Kd 12619

Erlang programok transzformációja bonyolultsági mérőszámok alapján

Doktori értekezés

2013

Király Roland

http://aries.ektf.hu/~serial/kiralyroland/

kiraly.roland@aries.ektf.hu

Témavezető: **Dr. Horváth Zoltán, egyetemi tanár**

Eötvös Loránd Tudományegyetem, Informatikai Kar

H-1117 Budapest, Pázmány Péter sétány 1/C.

ELTE IK Informatikai Doktori Iskola

Doktori program: Az informatika alapjai és módszertana

Az iskola és a program vezetője: Benczúr András egyetemi tanár

---

A KMOP-1.1.2-08/1-2008-0002 és a TÁMOP 4.2.4. A/2-11-1-2012-0001 - Jedlik Ányos Doktorjelölti Ösztöndíj a konvergencia régiókban pályázatok támogatásával.

Tartalomjegyzék

1. fejezet

Előszó

1.1. Bevezetés

A programok méretének és bonyolultságának növekedésével a fejlesztés (és a fejlesztési költségek) egyre nagyobb részét képezi a tesztelés. A tesztelés során felmerülő problémák megoldása, valamint a változtatást követően annak a bizonyítása, hogy a módosított programszöveg jelentése nem változott meg. A programok átalakításának a költségét nem kizárólag a méretük, hanem a forráskódjuk bonyolultsága is befolyásolja.

A bonyolultság mérése becsülhetővé teszi a tesztelési és karbantartási költségeket, ami fontos szempont a programfejlesztés során.

A különböző elemzések során a forrásszöveg azon tulajdonságait mérjük, amelyek segítségével képet kaphatunk annak karakterisztikájáról és fejlesztés során felhasznált program konstrukciók strukturális bonyolultságáról. Az így kapott eredmények alapján becsléseket adhatunk a programszöveg tesztelési, fejlesztési, valamint átalakítási költségeire [50].

A bonyolultságot mérhetjük a forráskód struktúrája [48, 49, 47, 45], valamint mérete alapján. Vizsgálhatjuk a programszöveget a fejlesztési fázisban, (*process metrics*), vagy a kész program felhasználhatósága alapján. Ez a fajta vizsgálat a végterméket jellemzi (*product metrics*), de szorosan kapcsolódik a forrásszöveghez és a modellhez, ami alapján a forrásszöveget felépítették.

A strukturális bonyolultságot mérhetjük a fejlesztési költségek (*cost metrics*), a szükséges erőfeszítés (*effort metrics*), vagy a fejlesztés előre haladottsága (*advancement*), esetleg a megbízhatóság alapján (*non-relialibity* (hibaszám)) is. Lehet mérni az újrahasznosítás mértékének (*reusable*) számszerűsítésével és mérhetjük a funkcionalitást *functionality*, vagy a használhatóságot is, de a bonyolultsági mértékek mindegyike az alábbi három fogalom köré csoportosul: méret, komplexitás, stílus.

A szoftver bonyolultsági mértékek külön-külön, vagy együtt mérve minősíthetik a programozói stílust, a programozás folyamatát, a felhasználhatóságot, a várható költségeket és a programok belső jellemzőit.

Természetesen a programfejlesztés során mindig arra törekszünk, hogy a mérhető felhasználhatósági mérték, az erőforrások használata és a program belső jellemzői összhangban legyenek.

Mindezek alapján megállapíthatjuk, hogy a programok jellemzéséhez nem elegendő egyetlen tulajdonság, vagy attribútum vizsgálata, sőt nem elég az összes általunk ismert mérték összegyűjtése és mérése.

A programszöveg egyes részei közti kapcsolatokhoz hasonlóan a mértékek közti kapcsolatok felderítése és azok egymásra hatásának a vizsgálata adhat csak pontos képet a mérés tárgyát képező szoftverről.

A dolgozatban erre a megállapításra alapozva építjük fel azt a modellt, amelyet a szoftver mértékek vizsgálatához és a program transzformációk hatásainak az elemzésére, - és azok alapján az automatikus, a program forrásszövegének minőségét javító transzformációk lebonyolításához - használhatunk fel.

A strukturális bonyolultságot a *product metrics* és a *process metrics* típusú mértékeket alapul véve, a forráskódból építhető összetett adattípust ($\mathcal{SG}$ lásd: 2. fejezetben) elemezve vizsgáljuk meg, valamint megmutatjuk azt, hogy a fejlesztési, tesztelési és átalakítási munkákat hogyan segítheti a bonyolultság mérése.

A szoftver bonyolultságból adódó problémák. Thomas J. McCabe [48, 34] már 1976-ban rámutatott arra, hogy milyen fontos a forráskód strukturájának az elemzése. McCabe cikkében leírja, hogy az IBM [51] által javasolt, a programkód kezelhetősége szempontjából ideális 50 soros modulok esetén 25 egymást követő `IF THEN ELSE` konstrukció is 33.5 millió különböző végrehajtási ágat eredményez. Ilyen nagyszámú végrehajtási ágat bármely ma ismert rendszer használata mellett sem lehet emberi időn belül tesztelni, ezáltal a program helyességét tesztelési módszerek használatával igazolni [35]. Az általa publikált bonyolultsági mérték segítségével tehát azt vizsgálta, hogyan lehet a programokat úgy modularizálni, hogy azok a legkisebb költséggel tesztelhetőek legyenek. Porkoláb Z. [34, 40] említi a McCabe ciklomatikus számot, mint az objektum orientált programok metódusainak egy lehetséges mérőszáma, melyet felhasznál a doktori értekezésében.

A probléma vizsgálata rámutat arra, hogy a programok bonyolultsága a forráskódban található vezérlő szerkezetek száma, azok egymásba ágyazottságának mértéke, és a forráskód minden egyéb mérhető attribútuma fontos szerepet játszik abban, hogy milyen költségekkel, és milyen mértékű költség ráfordítással lehet azokat tesztelni, javítani, átalakítani.

Nagyméretű, robusztus forrásszövegre a gyakorlatban alkalmazható technológia a modulok meghatározott (Erlang programoknál a *cluster* elnevezést használják [12]) méretű egységekbe sorolása, vagy a nagyméretű modulok szétbontása kisebb egységekre.

A modulok méretének optimálissá tétele a bennük szereplő programelemek csoportosításán alapszik. A programkódok elkülöníthető részeinek vizsgálata, vagy a programrészek kapcsolatainak a felderítése fontos feladat, mivel segítségével a modularizáció könnyedén elvégezhető.

A modulokra bontáshoz meg kell találni azokat az programelemeket (mivel funkcionális, Erlang [52], [32] forrásszövegről beszélünk, függvények és rekordok mentén végezzük el a vizsgálatot), melyek kapcsolatban állnak egymással. Egymást hívó, vagy közös adatokat használó függvények kapcsolatrendszerének felderítésével lehetőség nyílik a méretből adódó problémák megoldására.

A megfelelő transzformációk (*refactoring*) használatával és a szükséges kompenzációk bevezetésével új modulokba tudjuk csoportosítani, vagy a meglévő modulok között szét tudjuk osztani az összetartozó és a külön modulba tartozó függvényeket.

A kapcsolatok felderítéséhez, azok erősségének a meghatározásához, a megfelelő program transzformációk elvégzéséhez, valamint az elemzésre fordított idő csökkentésében szintén segítséget nyújthat a forráskód bonyolultsági mérőszámainak a vizsgálata.

1.2. Erlang programok bonyolultsága

Erlang. Az Erlang funkcionális programozási nyelv, amely konkurens, valós idejű, elosztott, nagy hibatűrésű rendszerek készítésére szolgál. Az Ericsson és az Ellemtel Computer Science Laboratories dolgozta ki és használja a mai napig telekommunikációs szoftverek készítésére.

Kidolgozását elsősorban az indokolta, hogy a 90-es évek elején nem volt a fejlesztők kezében az igényeiknek megfelelő programozási nyelv. A szekvenciális programozás támogatására széles körben ismertek voltak a C és Pascal nyelvek, de a valós idejű rendszerek készítői számára az Assembly volt az egyetlen eszköz.

A nyelvben elérhető konkurencia modell magas szintű támogatásának bevezetését elsősorban olyan problémák motiválták, ahol a probléma természetéből fakad a magas szintű párhuzamosítás (pl. valós idejű irányító rendszerek).

A funkcionális programozási nyelvek, így az Erlang nyelv is számos olyan különleges programkonstrukciót tartalmaz, amelyeket az objektum orientált és az imperatív nyelveknél nem találhatunk meg.

- Halmazkifejezések használata (Lista kifejezések)
- Lusta vagy mohó kifejezés kiértékelésének lehetősége
- Végrekurzió
- Változók egyszeri kötése

- Destruktív értékadás hiánya
- Ciklus típusú iterációk hiánya
- Számos esetben függvények hívási helyfüggetlensége
- Mintaillesztés
- Currying (Az Erlang nyelv csak a lehetőségét tartalmazza)
- Magasabbrendű függvények (λ (lambda) kifejezések használata)

A felsorolásban szereplő nyelvi elemek teszik a funkcionális nyelveket mássá, és ezektől a tulajdonságoktól válnak érdekessé, vagy különlegessé, de szintén ezek miatt az ismert bonyolultsági mértékek egy része nem, vagy csak átalakítással használható a programkódjuk mérésére.

Ez nem jelenti azt, hogy ezekhez a nyelvekhez nincsenek kidolgozva bonyolultsági mértékek, de kevés olyan van a meglévők között, amely általános felhasználású, vagyis bármely funkcionális nyelvre [43, 38, 37], így Erlang programok mérésére is alkalmazható (legtöbbjük speciálisan, csak az adott nyelvre mér jól, így Erlang forrásokhoz alacsony hatékonysággal használható).

Mindezért a bonyolultság méréséhez definiálnunk kell azokat a mértékeket, amelyeket ennél a paradigmánál alkalmazni tudunk, de készítenünk kell újakat is.

Meg kell tudnunk mondani azt is, hogy a program mely tulajdonságait kell vizsgálnunk. Mik a helyes értékek egy adott mérték esetében általában és az aktuálisan mért programra vonatkozóan is.

Az imperatív és objektum orientált programok mérésére alkalmazott mértékek [39], [44] közül sok kissé átalakítva, a funkcionális programok jellemzőihez igazítva használható az Erlang programszövegek mérésére.

Az imperatív nyelveknél megszokott *library* modulokban, vagy az objektum orientált programok névtereibe rendezett funkcionalitás ugyanúgy típus szerint összegyűjtött függvények (alprogramok, vagy metódusok) halmazát jelenti, mint az Erlang programok moduljaiba rendezett függvények esetében. Így az objektum orientált nyelvek osztályaiban a metódusokra alkalmazott mértékek [48, 46] némi átdolgozás után alkalmasak a funkcionális nyelvek függvényeinek a jellemzésére is.

Az Erlang nyelvben a függvényeket modulokba rendezzük. A függvény moduljait exportáljuk, hogy azok a modulon kívül is elérhetőek legyenek. Ez a módszer hasonlít az objektum orientált nyelvekben megismert osztályokra és láthatósági (*visibility*) szintjeire, mint a *private, public, protected.*

Az Erlang nyelvben a nem exportált függvények és az adatok lokálisak a modulra nézve, és a függvényekben definiált változók a függvényre.

Ha globális adatokat szeretnénk definiálni, akkor rekordokat készítünk, amelyeket fejléc fájlokban, vagy adatbázisokban helyezünk el.

Ez a struktúra, vagyis a modul és a benne definiált függvények tehát hasonlóak az osztályhoz és metódusaihoz. A globális változók a publikus mezőkhöz és az Erlang export listájának hatása az osztály publikus védelmi szintjeinek a hatásához.

Összefoglalva, a nem funkcionális nyelvekre alkalmazott mértékek közül azok használhatóak Erlang programok mérésére, amelyek az alprogramok tulajdonságait mérik, vagy az azok törzsében található vezérlő szerkezeteket és adatokat, esetleg az alprogramok közti kapcsolatokat, mint a függvény hívás, vagy az adatfolyam.

A szoftverbonyolultság méréséhez találunk eszközöket, mint az Eclipse [53], vagy a Simon, Steinbrückner és Lewerentz [41] által készített, számos bonyolultsági mérőszámot implementáló szoftver, amely segíti a felhasználót a mérések kivitelezésében.

A *Crocodile* [42] projekt célja egy olyan program megalkotása, amely segíti a hatékony forráskód elemzést, így jól használható forráskód transzformációk utáni mérések lebonyolítására.

A *Tidier* [13, 14] egy automatikus forráskód elemző és transzformáló eszköz, amely képes automatikusan javítani a forrásszöveget, és kiküszöbölni annak az egyszerű forrásszöveg elemzés segítségével felderíthető hibáit. Viszont egyik szoftver vagy alkalmazott módszer sem használ bonyolultsági mérőszámokat a forrásszöveg elemzése során.

A célunk tehát, hogy kidolgozzunk egy Erlang programok mérésére alkalmas, komplex mértékrendszert, amely segítségünkre lesz a dolgozatban bemutatott, illetve a tézisekben leírt eredmények megvalósításához.

Meg kell találnunk továbbá azokat a módszereket, amelyekkel az iparban is használatos, nagy méretű, elosztott programozást és viselkedési minták alkalmazását is támogató Erlang programok bonyolultsága mérhetővé, a forrásszövegük gyorsan és egyszerűen javíthatóvá válik.

Meg kívánjuk mutatni azt is, hogy a használt mértékek megváltozása tükrözi a forrásszöveg változásait mind a program transzformációk (*refactoring*), mind a forráskód egyéb változtatása esetén. Viszont ez fordítva is igaz, vagyis a program változásai nyomán a bonyolultsági mértékek megfelelően változnak, tehát mérik a program transzformációk hatásait.

Mindezek mellett ismertetjük azt a forráskód statikus analízisével felépíthető szemantikus gráfot [23] (részletesen a 2. fejezetben), amely segítségével a strukturális bonyolultsági mértékeket mérjük, valamint azt az elemző és transzformációs algoritmust, ami lehetőséget biztosít számunkra a forrásszöveg bonyolultsági mértékek alapján történő optimalizálására.

A dolgozatban az alábbi eredményeket, és azok elméleti hátterét, valamint az eredmények gyakorlati alkalmazásának módszereit ismerhetjük meg:

- Az Erlang forráskód javításra használható transzformációs lépéseket, valamint azok hatásait a forrásszöveg transzformáció által közvetlenül, valamint közve-

tetten érintett részein. A funkcionális nyelvekre átdolgozott, valamint az Erlang nyelvhez kidolgozott szoftver bonyolultsági mértékeket és alkalmazásaikat.

- A bonyolultsági mértékek lekérdezésére kifejlesztett strukturált lekérdező nyelvet, és annak szűrőit, amely nyelv segítségével az Erlang programok bonyolultsága egyszerűen és gyorsan mérhető.
- Egy olyan bonyolultság alapú elemző algoritmust, amely segítségével a programkészítés bármely fázisában, a bonyolultsági mértékek változása alapján felismerhetővé válik számos programfejlesztési hiba, vagy következetlenség. Az algoritmus segítségével előre definiált (a fejlesztő által előírt, paraméterként megadható), jól meghatározott keretek közé szorítható a programfejlesztés.
- A lekérdező nyelv automatikus programtranszformációk kivitelezésére használható kiterjesztését (transzformációs metanyelv), valamint a nyelv felhasználásával készült szkriptek futtatására alkalmas algoritmust és módszert, amely segítségével a bonyolultsági mérték alapú automatikus forráskód transzformálás és javítás megvalósíthatóvá válik.

A dolgozatban bemutatásra kerülő módszerek és algoritmusok részét képezik az ELTE IKKK, a KMOP-1.1.2-08/1-2008-0002 és az Ericsson Magyarország által támogatott projekt keretein belül készített *RefactorErl* [33] nevű programnak, amely jelenleg is fejlesztés alatt álló, és az iparban prototípusként működő szoftverrendszer.

RefactorErl. A *RefactorErl* egy forráskód elemzésére és transzformációjára alkalmas szoftver, amely Erlang programok fejlesztésére és a forrásszöveg transzformálására alkalmas. A program transzformációs lépések szabályokhoz vannak kötve, vagyis kizárólag olyan transzformációk végezhetőek el, amelyek az adott lépéshez definiált szabályoknak megfelelnek és a program eredeti jelentését nem változtatják meg.

A programszöveg feldolgozásához saját szintaktikai, szemantikai és lexikális elemzővel rendelkezik, amely alkalmassá teszi az olyan feladatok elvégzését is, amikre az eredeti fordítóprogram által készített szintaxisfa (Abstract Syntax Tree) használata mellett nem lenne képes. A szoftver nem közvetlenül a forrásszövegen hajtja végre a különböző transzformációkat, hanem az abból épített speciális, szemantikus információkkal kiegészített szintaxisfát transzformálja, majd az átalakítást követően újra előállítja a forrásszöveget. A visszaállítás során a lehető legtöbb helyen megőrzi az eredeti *layout*-ot, vagyis a programozó által használt elrendezését a nyelvi (lexikális) elemeknek.

A programban elérhető 24 transzformációs lépés egyszeri, vagy sorozatos elvégzése a programok jelentését nem változtatja meg, de átláthatóbbá, kezelhetőbbé teszi a forráskódot.

Jellegéből és funkciójából adódóan olyan forrásszövegeket is képes beolvasni, amelyek olyan hibákat tartalmaznak, amiket az általános felhasználású fordítóprogramok

nem engednek meg. A hibákat az elemzések és átalakítások során megőrzi mindaddig, amíg azokat a programozó meg nem szünteti. A szintaktikai hibás programelemeket is képes beolvasni a tárolásra használt adatstruktúrába, majd vissza konvertálni a forrásszövegbe, lehetővé téve, hogy a program más részeinek javítása és átalakítása közben azok jelenléte ellenére dolgozhassunk a forrásszöveg más részeivel.

A tézisekben bemutatott strukturális bonyolultságot mérő elemző algoritmus, a szöveges lekérdező nyelv és annak kiterjesztése, a szkriptek futtatását végző automatikus transzformációs algoritmus, az automatikus hibajavító módszerek és a kódbonyolultság alapú hibadetektálás mind új eredmények, amelyek a *RefactorErlben* kerültek implementálásra. A tézisek mindegyike publikálásra került az irodalomjegyzékben felsorolt és a fejezetekben megjelölt szakfolyóiratokban, valamint konferencia előadásokon.

1.3. A dolgozat felépítése

Dolgozatban tehát bemutatjuk azt a bonyolultsági mértékeken alapuló elemző és optimalizáló algoritmust, amelyet *Erlang* nyelvű forrásszövegek készítése során, valamint korábban készült, de átalakításra váró programszövegek automatikus, vagy félautomatikus javítására használhatunk.

Az *Erlang forrásszöveg gráf reprezentációja és bonyolultságának mérése* című fejezetben Az Erlang nyelvű forrásszöveg tárolására kidolgozott struktúrát, az alkalmazott bonyolultsági mérőszámokat, valamint a bonyolultság mérésének lehetőségeit vizsgáljuk.

Az *Erlang forrásszöveg gráf reprezentációja bonyolultság méréséhez* című szakaszban megismerhetjük azt az adatstruktúrát, amelyet a RefactorErl [23] szemantikus gráfjának továbbfejlesztésével a forrásszöveg bonyolultságának méréséhez és tárolásához készítettünk, amely tartalmazza az adott program minden, számunkra fontos, statikus analízissel mérhető attribútumát, a program hívási gráfját, valamint az adatfolyam gráfot [22] és még számos olyan információt, amely a bonyolultságot elemző algoritmus működéséhez szükséges.

Az *Erlang programok bonyolultsági mérőszámai* című részben bemutatásra kerülnek azok a strukturális bonyolultsági mértékek, amelyeket az Erlang forrásszövegek bonyolultságának méréséhez állítottunk össze már meglévő mértékek átalakításával és újak kidolgozásával.

A *Szöveges lekérdező nyelv* című szakaszban megvizsgáljuk a bonyolultsági mértékek kiszámításához kidolgozott strukturált lekérdező nyelvet, amely használata lehetővé teszi a bonyolultsági mérték mérését a programszöveget tároló adatstruktúra és a bonyolultságot elemző algoritmus ismerete nélkül. A fejezetben bemutatjuk a nyelv szintaxisát, valamint példákon keresztül a használatában rejlő lehetőségeket.

Ez a fejezet az ***Erlang nyelvre alkalmazható metrikák kidolgozása, mérése és a lekérdező nyelv megalkotása*** tézis alapjául szolgál.

A *bonyolult programrészek lokalizálása* című fejezetben ismertetjük, hogyan lehet az általunk kifejlesztett elemző algoritmus használatával a forrásszövegen mért szoftver bonyolultsági mértékek eredményei alapján lokalizálni és megjelölni a kezelhetetlenül bonyolult programrészeket. Ebben a szakaszban megvizsgáljuk azt is, hogy a különböző programtranszformációk hogyan hatnak a kódminőségre és milyen irányba változtatják meg programokat jellemző bonyolultsági mértékeket.

Bemutatjuk a bonyolultság alapú elemző algoritmus működési elvét és az elemzések elvégzésére kidolgozott szabályrendszert, ami segít abban, hogy megtaláljuk azokat a programfejlesztési hibákat, amelyeket a bonyolultsági mértékek változásai alapján detektálni lehet.

A fejezet a ***Bonyolult programrészek lokalizálása és kódjavítást célzó programtranszformációk hatása a kódminőségre*** című tézist, és annak eredményeit ismerteti.

Az *Automatikus program transzformációk* című fejezetben megvizsgáljuk annak a lehetőségét, hogy a bonyolultsági mértékek mérésével, valamint az elemző és hibadetektáló algoritmus alkalmazásával hogyan lehet a bonyolultsági mértékek automatizált javítását célzó programtranszformációkat megvalósítani. Ez a fejezet az elméleti háttér, és a módszerek bemutatása mellett ismerteti az automatizált transzformációs lépések gyakorlati alkalmazásának lehetőségeit is.

A fejezet a ***Forráskód automatikus transzformációja szoftver bonyolultsági mértékek alapján, és kódminőség javító transzformációs sémák kidolgozása*** című tézis eredményeit ismerteti és kiegészíti az első tézisben bemutatott szöveges lekérdező nyelvet olyan elemekkel, amelyek a bonyolultság mérése mellett lehetővé teszik a bonyolultsági mértékeken alapuló, automatikus transzformációk készítését.

Az így kapott transzformációs metanyelv segítségével olyan szkripteket készíthetünk, amelyekkel a fejlesztés alatt álló, vagy korábban már elkészített, de kézzel kezelhetetlen méretű forrásszöveg bonyolultsági mértékek alapján detektálható hibái automatikusan javíthatóvá válnak. A nyelv ismertetésével párhuzamosan megismerhetjük azt az elemző és transzformáló algoritmust is, amely lehetővé teszi a kiterjesztett lekérdező nyelven írt szkriptek futtatását. A fejezet végén példaprogramok és futási eredményeik bemutatásával igazoljuk az automatikus hibajavítás működőképességét.

2. fejezet

Erlang forrásszöveg gráf reprezentációja és bonyolultságának mérése

2.1. Erlang forrásszöveg gráf reprezentációja a bonyolultság méréséhez

2.1.1. Bevezetés

Nyékiné és Fóthi [44] az adatfolyamok nyomon követéséhez az AV gráfot, valamint Porkoláb [34, 40] az AV gráf alapján az objektum orientált programok bonyolultságának méréséhez egy speciális gráfot definiál, amely segítségével a programokat jellemző bonyolultsági mértékeket kiszámítja. Mi ezt a módszert alapul véve a programgráfhoz hozzárendeljük a szintén statikus analízissel nyerhető hívási és adatfolyam gráfot, valamint a programszöveget jellemző bonyolultsági mérőszámokat. Bemutatjuk a gráf bejárásához készített útvonal kifejezések működését és az útvonal kifejezések alkalmazásához konstruált elemző algoritmust.

2.1.2. A fejezetben ismertetésre kerülő eredmények

A fejezetben az AV gráfot, valamint a RefactorErl [23] szemantikus gráfját egészítjük ki úgy, hogy az képes legyen a program konstrukciók mellett a szoftver bonyolultság kiszámításához szükséges attribútumok tárolására.

A kiegészített adatstruktúrához tartozó elemző algoritmust kiterjesztjük úgy, hogy az alkalmassá váljon a szoftver bonyolultsági mérésére, elemzésére és a bonyolult programrészek jelölésére a programszöveget reprezentáló gráfban.

Bemutatjuk továbbá a kiterjesztett gráfon értelmezhető útvonal kifejezéseket leíró nyelv szintaxisát és élcímkéinek informális szemantikáját. Ezt a nyelvet szintén a

RefactorErl alacsony szintű gráf bejáró nyelvének továbbfejlesztésével hoztunk létre. Végül megmutatjuk az útvonal kifejezések futtatásához konstruált elemzőt, és annak működését.

Végül ismertetjük azt, hogy az útvonal kifejezések és az elemző algoritmus segítségével hogyan lehet a szoftver bonyolultsági mértékek kiszámításához szükséges speciális gráf útvonalakat leírni és futtatni.

2.1.3. Programelemek tárolása és elemzése

Az alábbiakban matematikai eszközök segítségével definiáljuk az általunk használt $\mathcal{SG}$ szemantikus gráfot.

A fordítóprogramokhoz hasonlóan a RefactorErl elemző algoritmusa elsőként a forrásszöveg szintaktikai struktúráját reprezentáló szintaxisfát építi fel, majd kiegészíti a szemantikus információt gyűjtő függvények [23] által szerzett információkkal. Az elemző ezen a ponton nem hagyja abba a munkát, hanem a szemantikus információval kiegészített gráf csomópontjaihoz hozzárendeli az azokhoz kapcsolódó lexikális elemeket, *white space* karaktereket, kommenteket, valamint minden olyan szöveges információt, amit a modulokban talál.

Erre azért van szükség, mert a forrásszöveget érintő transzformációkat a szemantikus gráfon kell elvégezni (a gráfot transzformálja), és a változtatásokat követően a programelemeket reprezentáló gráf csomópontokat információ vesztés nélkül vissza kell alakítani forrásszöveggé (a fordítóprogramok elemzői számos lexikális elemet eldobnak az elemzések során).

Az így szerzett információ tárolására használt G szemantikus gráf egy absztrakt adattípus, amely reprezentálja a forrásszöveg szintaktikai és szemantikai struktúráját, vagyis a program minden egyes, jól elkülöníthető részegységének a tulajdonságait, a programrészek közötti kapcsolatokat, a hívási és egyéb útvonalakat, valamint a lexikális elemeket. Az adattípus egy irányított élcímkézett gráf, ami a következő adatstruktúrával jellemezhető [23, 22]:

2.1. definíció. (*Szemantikus gráf*)

$$\mathcal{G} = (N, A_N, A_V, A, T, E)$$

szemantikus gráf, ahol az N gráf csúcsok halmaza. Az A_N a csúcsok attribútum neveinek a halmaza, A_V a tulajdonságok (attribútumok) értékei, A a csomópontokat osztályozó függvény, T az irányított élek halmaza (éltípusok) és E parciális leképező függvény, amely a csomópontokat összekötő éleket definiál a gráfban.

(A gráf elemeinek bővebb kifejtése a 2. definícióban és a hozzá tartozó magyarázatban található).

Az általunk a dolgozatban használt, és az eredeti $\mathcal{G}$ gráf kiterjesztésével létrehozott $\mathcal{SG}$ gráf úgy áll elő, hogy bevezetjük M-et, ami egy speciális gráf csúcs a bonyolultsági mértékek méréséhez. Ekkor $\mathcal{SG}$ a következő formában írható le:

2.2. definíció. ***(Kiterjesztett szemantikus gráf)***

$$\mathcal{SG} = (N, A_N, A_V, A, T, E, M),$$

ahol

- N *a gráf csúcsok halmaza, mely csúcsok reprezentálják a forrásszöveg szintaktikus, szemantikus és lexikális elemeit.*
- A_N *az* $n \in N$ *csúcsok tulajdonságainak (attribútum neveinek) a halmaza,*
- A_V *a lehetséges tulajdonságok értékei (az adott csúcsot jellemző adatok),*
- $A : N \times A_N \to A_V$ *a csomópontokat osztályozó parciális függvény*
- T *irányított élek halmaza (éltípusok), mely élek a különböző csomópontok kapcsolatait adják az* $\mathcal{SG}$ *szemantikus gráfban (minden élcímke egy éltípust jelöl a gráfban, amely egy jól meghatározott csomópont típusba tarthat).*
- $E : N \times T \times \mathbb{N}_0 \to N$ *parciális leképező függvény, amely a szemantikus gráf csomópontokat utakat (éleket) definiál a gráfban (egy csúcstípusból egy másikba vezető egy fajta élet).*
- $M = (t, M_A)$ *egy speciális csúcs. M-et az $\mathcal{SG}$ gráf N csúcsaihoz kapcsolhatjuk, és a kapcsolt (program elemeket reprezentáló) csúcsot jellemző bonyolultsági mértékeket tárolja.*

  A t a kapcsolt (programelem) csúcs típusát írja le (pl.: modul, függvény) és M_A rendezett hármasok halmaza, ahol a hármasok első eleme az adott (kapcsolt) programelemen mérhető bonyolultsági mértékek neve, a második a megnevezett bonyolultsági mérték elvárt értéke erre csúcsra nézve, a harmadik elem pedig a mérték aktuálisan mért értéke.

2.3. magyarázat. *Az M bevezetésével az $\mathcal{SG}$ gráfot egy olyan számunkra értékes tulajdonsággal ruházzuk fel, ami segítségünkre lesz a negyedik tézis megalkotásában (erről bővebben a 3. fejezetben olvashatunk). Ez a tulajdonság lehetővé teszi, hogy könnyedén megtalálhassuk az $\mathcal{SG}$ olyan pontjait, ahol a csúcson mért valamelyik (akár az összes) bonyolultsági mérték értéke eltér az elvárt értéktől. Az $\mathcal{SG}$ gráf és az elemző algoritmus a bonyolultsági mértékek kiszámítása során az elvárt, és a mért érték tárolásával mintegy megjelöli a gráf (így a gráf által reprezentált) forrásszöveg bonyolultság*

szempontjából problémás részeit. Ekkor nyilván az elemző algoritmust is ki kell egészítenünk egy újabb elemzési menettel, ami a szemantikus elemzők lefutását követően a bonyolultságot is megméri a gráf csúcsokra, majd eltárolja azokat a gráfban.

2.4. megjegyzés. *Annak ellenére, hogy az M formálisan az $\mathcal{SG}$ része, ezeknek a csúcsoknak a gráfja nem az $\mathcal{SG}$ többi csúcsával tárolódik. Az elemző algoritmus egy olyan gráfot tart nyilván, amit az $\mathcal{SG}$ többi elemét tároló gráffal konzisztensen tart és összekapcsolja az M csúcsokat az $\mathcal{SG}$ megfelelő csúcsaival. Erre azért van szükség, hogy az algoritmus elemzési sebességét növeljük, és azért is, hogy a bonyolultság elemzését ki lehessen kapcsolni. (A bonyolultság elemző újbóli bekapcsolása nagy számításigényű folyamat, de csak egyszer kell elvégezni ahhoz, hogy az M csúcsokat tartalmazó gráf újra konzisztens legyen az $\mathcal{SG}$-vel.)*

2.5. megjegyzés. *Az M egy eleme, vagyis a bonyolultsági mértékeket tároló csúcstípus egy lehetséges példánya*

$$m_i = (func, [\{max\_depth\_of\_cases, 3, 6\}, \{lines\_of\_code, [2, 40,], 10\}, ...]).$$

A példában látható, hogy az adott függvény típusú csomópontra vonatkozóan a case kifejezések beágyazottságának az elvárt értéke három, de a függvényre mérve az érték hat, vagyis a programszöveget reprezentáló csúcs bonyolultsága nem megfelelő. Ezen a ponton valamely jelentés megőrző transzformáció segítségével javítani kell a forráskód minőségén (erről bővebben a 3. fejezetben olvashatunk). A második bonyolultsági mérték azért érdekes, mert ennél az elvárt érték nem egy konstans, hanem egy intervallum, amelynek minimuma és maximuma között kell lennie a mért eredménynek ahhoz, hogy az elemző algoritmus a bonyolultságot megfelelőnek ítélje.

Az $\mathcal{SG}$ elemzése során bejárt, élcímkékkel meghatározott utakat útvonal kifejezésnek nevezzük [23, 22]. Az útvonal kifejezéseket a csomópontok közötti kapcsolatokat reprezentáló élcímkékkel és az éleken való haladás irányával adjuk meg.

Ez azért fontos a számunkra, mert az útvonal kifejezések által definiált részgráfok bejárásával, és a bejárás során szerzett információ alapján határozzuk meg a különböző a 2.1.7. fejezetben bemutatott bonyolultsági mértékeket.

Minden útvonal egy speciális csúcsból, a gyökérelemből (*root*) indul és egy $n \in N$ csomópontba tart. Mindezek alapján a különböző bonyolultsági mértékek kiszámításához útvonal kifejezéseket kell konstruálnunk (minden mértékhez különböző utakat), hogy a bennük szereplő utak bejárása során általában gráf csúcsok halmazát, vagy numerikus értékeket kapjunk eredményül (részletesen lásd: a 2.1.7. fejezetben).

Az A leképező függvény adja meg az adott csomópont attribútum értékeit. Ahol az n csomópont attribútuma a, és az attribútum értéke $A(n, a)$. Egy tetszőleges $n \in N$ csomópontból kiinduló élek $t \in T$ tagjának az iránya fixált, és az i. él végpontja a

következő formában adott $E(n,t,i)$. Az $n \in N$ a kiindulási csomópont a szemantikus gráfban, a $t \in T$ a csomópontból kiinduló él, vagy élsorozat, és az $i \in \mathbb{N}_0$ az n csomópontból induló, azonos címkéjű, sorszámozott élek esetén az él sorszáma. Az élek egy csomópontból kiinduló több azonos címkéjű él esetén (a 0 kezdőértéktől indulva) számozással különböztethetőek meg egymástól.

2.6. megjegyzés. *Így a magasabb szintű információ megszerzése a gráfból az alacsony szintű lekérdező nyelv (útvonal kifejezések) felhasználásával lehetséges, amely a gráf struktúra bejárását teszi lehetővé jól meghatározott mélységben. (Ez a lekérdező nyelv nem azonos a 2.2.5. fejezetben bemutatásra kerülő magasabb szintű, strukturális bonyolultságot lekérdező nyelvvel). A bonyolultság lekérdezésére használható, magas szintű nyelv a következő definícióban bemutatott útvonal kifejezéseken alapszik.*

2.7. definíció. (***Útvonal kifejezés***)

$$\begin{aligned} P &= [Pe_1, Pe_2, \ldots, Pe_k], \\ Pe_i &= (t_i, d_i, f_i), \end{aligned}$$

kifejezés egy P útvonalat ír le PE_i elemekkel (a gráfban az útvonalat reprezentáló élek típusai), ahol

- $t_i \in T$ *az útvonal két csomópont közötti útvonal leírására alkalmas él típusa,*
- $d_i \in \{forward, back\}$ *az adott élen értelmezett irány megadására szolgál, ahol a $forward$ előre, back visszafelé mutat,*
- $f_i : \mathbb{N}_0 \times N \to \mathbb{L}$ *az útvonal specializálására alkalmas függvény (az útvonal bejárásával kapott csomópontokat szűri azok attribútum értékei alapján).*

2.8. megjegyzés. *A bonyolultság mérésére általunk alkalmazott útvonal kifejezések sémáját szintén a RefactorErl szemantikus gráf útvonal [23, 22] kifejezéseinek a kiterjesztésével alkottuk meg, megtartva annak előnyeit, de kiegészítve a bonyolultság szempontjából fontos elemekkel. Az útvonalak kiértékelését végző függvénnyel ugyanígy jártunk el. Ezzel a módszerrel alkalmassá tettük az eredeti adatszerkezetet és a hozzá tartozó elemző algoritmust a szoftver bonyolultsági mértékek mérésére és tárolására. (az útvonal kifejezések szintaxisának leírása a 2.1 listában látható).*

Az útvonal kifejezések kiértékeléséhez bevezetjük az $\mathcal{E}$ függvényt [23, 22].

2.9. definíció. ($\mathcal{E}$ kiértékelő függvény)

$$\begin{aligned}
\mathcal{E}(n,[]) &= [n] \\
\mathcal{E}(n,[Pe_1,Pe_2,\ldots,Pe_k]) &= \mathcal{E}(\mathcal{E}(n,Pe_1),[Pe_2,\ldots,Pe_k]) \\
\mathcal{E}([n_1,\ldots,n_k],P) &= \mathcal{E}(n_1,P)+\ldots+\mathcal{E}(n_k,P) \\
\mathcal{E}(n,[(t,forward,f)]) &= \{n' \mid n' = E(n,t,i) \wedge f(i,n') = true\} \\
\mathcal{E}(n,[(t,back,f)]) &= \{n' \mid E(n',t,i) = n \wedge f(0,n') = true\}
\end{aligned}$$

2.10. magyarázat. *A gráfbejárások során az $\mathcal{E}(n,[])$ a függvény az $n \in N$ csúcsra és üres útvonal kifejezésre az n csúcsot adja eredményül lista ($[n]$) formában. Az $n \in N$ csúcsra és $[Pe_1,Pe_2,\ldots,Pe_k]$ a kiindulási csúcsból a Pe_1 élen vezető úton a következő n' csúcsot kapjuk, majd ebből az n'' csúcsot, és így tovább (szekvencia).*

Több $[n_1,\ldots,n_k]$ csúcs és a hozzájuk tartozó P élsorozat esetén minden csomópontból kiindulva, az utak bejárásának eredményeit kapjuk. A + operátor ebben az esetben nem összeadást, hanem konkatenációt jelent, ami az eredmény listákat fűzi össze.

A $(t,forward,f)$, és a $(t,back,f)$ $Pe_i \in P$ éleket leíró rendezett hármasokban a t csomópontból indulva az élen előre, vagy visszafelé haladást definiálhatjuk úgy, hogy megadjuk az él típusát, az irányt és a szűrő feltételt, vagyis az él specializációját.

A kifejezés eredménye általában $n \in N$ csomópontok halmaza, de lehetséges a csomópontok halmazának a halmaza, vagyis a több csomópontra alkalmazott $\mathcal{E}$ függvények eredményeinek a halmaza: $\mathcal{E}(n_1,P)+\ldots+\mathcal{E}(n_k,P)$.□

A szemantikus gráf bejárására alkalmazott $\mathcal{E}$ függvény segítségével lehetőségünk nyílik a 2.1.7. fejezetben található strukturális bonyolultsági mértékek lekérdezésére.

Az adott mérték eredményének kiszámításához meg kell határoznunk a (bejárandó) P útvonalat a szemantikus gráfban, vagyis definiálnunk kell a megfelelő útvonal kifejezést, majd az útvonal bejárása (az $\mathcal{E}$ kiértékelő függvény alkalmazásával) során minden egyes lépésben el kell végeznünk különböző számítási lépéseket.

Az elemi lépés ebben az esetben egy csomópontból egy másik csomópontba való eljutást, az összetett lépés az él sorozatokon, valamint az így érintett gráf csúcsokon való áthaladást, majd a kapott eredmény kiértékelését jelenti.

Az elemzések során általában megszámoljuk az élek címkéivel, és a csúcsok attribútum értékeivel definiált útvonalon elérhető csúcsok előfordulásait, máskor meg az útvonal hosszát, vagy a kijelölt részgráf mélységét mérjük, de mindig a gráf útvonalak bejárásakor elvégzett számolási műveletekkel jutunk azokhoz a részeredményekhez, amelyekből a későbbiekben kiszámolhatjuk az adott bonyolultsági mérőszámot.

2.1.4. Gráf élcímkék informális szemantikája

Az $\mathcal{SG}$ gráf a programelemeket reprezentáló csúcsokból épül fel (node). Egy adott csúcshoz tartozhatnak a belőle kiinduló és az ide érkező irányított élek, valamint olyan attribútumok, amelyek egyedivé teszik.

Az irányított éleket két tulajdonság jellemzi. Az egyik az, hogy milyen típusú szemantikus gráf csúcsokat kötnek össze, a másik pedig az élcímke, amely a csúcsok közti kapcsolat típusát jelöli.

Attribútum pl. a *name*, amely számos csúcstípushoz azonosítót rendel, és címkézett, irányított élre példa a fájl és a modul közti kapcsolatot jelképező *moddef* címkéjű él.

A különböző csúcsokat azonos címkéjű élek is összekapcsolhatják és egy csúcshoz többféle irányított él is kapcsolódhat. Az útvonal kifejezések megadásánál az él irányát kétféleképpen adhatjuk meg. A címke neve az "előre", a címke neve és a *back* módosító a "visszafelé" haladást szimbolizálja (pl. *modul*, és *{module, back}*).

A gráfon értelmezett útvonalak definiálása tehát élcímkék megadásával történik. Az utakat ez alapján úgy írhatjuk le, hogy megadjuk a csúcsokat összekötő élek, élsorozatok élcímkéit (neveit), és azokat a szűrő feltételeket, amelyek az utakat specializálják.

Az élcímkék megadásánál nem törekszünk arra, hogy minden éltípust bemutassunk, helyette azokat az éleket ismertetjük, amelyek a bonyolultsági mértékek kiszámításánál elengedhetetlenek, és a dolgozatban felhasználunk.

Speciális gráf csúcs a *root*, ami a gráf gyökér eleme és minden elemzett programszöveghez egy létezhet belőle. Minden egyes lekérdezés (gráfon értelmezett útvonal) a *root*-ból indul, és innen vezet el a keresett részgráf kezdő, valamint végpontjához.

Kifejezések. A kifejezéseket más csúcsokhoz kapcsoló élek a *expr, sub_expr, top, visib, eattr, esub, varbind, varref, recref, fieldref, modref, funlref, funrref, dynfunlref, dynfunrref*, amelyek közül a számunkra érdekesek a *sub_expr, sub, esub*. Ezek a kifejezést más kifejezésekhez kötik alá és felé rendeltségi viszonyban (pl. részkifejezés, vagy beágyazott kifejezés), valamint a *funref, dynfunref, funrref* és a *funlref*, amelyek a függvényekhez kapcsolják a kifejezéseket.

A kifejezéseket jellemző attribútumok a *type*, amely a kifejezés típusát jelöli és a *value*, amely az értékét definiálja.

Kontextus információ. Egy adott programelem környezetéről kaphatunk információt a *top, visib, scope, clause* címkéket használva. A *top* címke az egymással összekapcsolt kifejezések közül a legfelső szintű (top-level) kifejezéseket jelöli. A *scope* címkével jelölt kapcsolaton keresztül a kifejezést tartalmazó klózokat (clause) érhetjük el, a *clause* címke pedig az adott programelemet közvetlenül tartalmazó klózhoz vezet. A *visib* a legfelső szintű kifejezéseket adja a klózokban.

csúcstípus	*attribútumok*	*élcímkék*
kifejezés	type, value	expr, visib, sub_expr, top, eattr, sub, esub, varbind, varref recref, fieldref, modref, funref, funlref, funrref, dynfunlref, dynfunrref
kontextus		top, visib, scope, clause
file	type, path, eol, lastmod	incl, form, record
form	type, hash, form_length, start_scalar	parent, tag, pp
modul	name	module, file, moddef, modctx, scope, modref, metric
függvény	name, arity, dirty, regular, opaque	func, fundef, funref, funexp, funimp, funcall, metric
változó	name	variable, varvis, varintro, varbind, varref, scope
rekord	name	recdef, field, recref
makró	name	mref
lexikális elemek		elec, clex, tlex, flex, start, next, last, stop

2.1. táblázat. Az $\mathcal{SG}$ gráf csúcs típusait összekötő élek, és a csúcsokat jellemző attribútumok

Modulok. A modulokhoz számos különböző információt tartalmazó szemantikus csúcsot rendelhetünk. Minden modul a *module* címkével kapcsolódik a *root* csúcshoz és a modult tartalmazó *file* csúcshoz a *moddef* élen keresztül jutunk el. A *modctx* az összes *scope*-ba tartozó klózhoz vezet és a *modref* élek minden a modulhoz explicit kapcsolódó kifejezéshez vezetnek. A modul csúcsokhoz a *metric* élekkel a bonyolultság lekérdezésének értékeit, valamit az elvárt értékeket tároló szemantikus csúcsokat köthetjük hozzá. A *name* az adott modul nevét tartalmazó attribútum.

Függvények. A függvényekhez rendelhető attribútumok a *name, arity, dirty, regular* és az *opaque*. Az első kettő az adott függvény nevét és paramétereinek számát írja le. A maradék három attribútumnak a szoftver bonyolultság mérése szempontjából nincs jelentősége .

A függvény csúcsokhoz kapcsolódó élek címkéi közül a *func* a függvényt definiáló modulhoz vezet (oda, ahol a függvény definíciója szerepel). Magát a függvény definíciót a *fundef* élen keresztül érjük el. A *funexp* élet akkor használjuk, ha egy függvényt egy modul exportál, és ekkor a exportot reprezentáló csúcshoz juthatunk el rajta keresztül. A *funimp* élcímke az importált függvény és az importáló modul közötti kapcsolatot reprezentálja. A függvényre explicit hivatkozó kifejezések a *funref* élen, a *funcall* éleken pedig a függvényeket hívó más függvények szemantikus csúcsit érhetjük el. Ahogy azt a modul csúcsoknál láthattuk, a függvény típusú csúcsokhoz is kapcsolhatunk *metric* típusú élekkel az adott függvény bonyolultságát leíró szemantikus csúcsokat.

Változók. A változókhoz is különböző szemantikus csúcsokat és attribútumokat adhatunk. A *name* attribútum a változó egyedi azonosítóját tartalmazza. A *variable, varvis, varintro, varbind, varref* éleken keresztül pedig számos szemantikus csúcshoz juthatunk el.

A változó *scope*-jába (élettartam) tartozó klóz, a *variable* élen, és az összes klóz, amelyekben a változó látható *varvis* éleken érhető el. A legfelső szintű kifejezést, amely a változót bevezeti (a "definíció" helye) a *varintro* élen kapjuk meg, és azok a kifejezések, ahol a változóhoz kötjük annak értékét a *varbind* élen keresztül. A *varref* élek az összes olyan kifejezéshez vezetnek, amelyek a változóra explicit módon hivatkoznak.

Rekordok. A bonyolultság mérése szempontjából rekordokhoz kötött információk közül a *name* attribútum, és a *recdef* élen elérhető rekord definíció, valamint a rekordok mezőihez vezető *field* élek az érdekesek. A *recref* azokat a csúcsokat (kifejezéseket) jelöli, ahol a rekordra hivatkozás található.

Formok. A modulok formokból állnak össze, amelyekhez a *type, parent, pp* élek és a *form_length, start_scalar, start_line* attribútumok kapcsolódnak, melyek közül a *type* az adott form típusát jelöli, a *parent* él pedig a szülő csúcshoz vezet el a gráfban.

Makrók. A makrók esetében az egyetlen számunkra fontos attribútum a *name*, amely a makrót azonosítja. Az egyetlen él, amit a bonyolultság kiszámításakor használunk az *mref*, amely a makró felhasználásának helyét azonosítja.

Fájlok. A fájl típusú csúcsokhoz a *type* attribútum a fájl típusát rendeli hozzá, a *path* az elérési útját a fájlrendszerben, az *eol* a fájlvége jelet, a *lastmod* pedig értelem szerűen az utolsó módosítás időbélyegét tartalmazza. Az *incl* élen azokat a fájl csúcsokat kapjuk, amelyeket fejléc fájlként használunk egy adott fájlban, és az *incl* címkét a *back* módosítóval specializálva (*{incl, back}*) azokat a fájlokat kapjuk, amelyekben az adott fájlt használják fejléc fájlként. A *form* élen a fájl moduljában található formokat, a *record* élekkel az itt definiált rekordokat érjük el.

A *type* típusú attribútumok számos csúcstípusban megtalálhatóak, úgy mint a kifejezések, formok, fájlok, mivel ez az információ az elemzéseket tekintve nagyon fontos és jó, ha gyorsan el lehet érni, vagyis gyorsan el lehet dönteni a programelem speciális típusát (újabb útvonal alapú elemzések nélkül).

Hasonló a helyzet az élcímkékkel. A különböző csúcstípusokhoz köthető azonos élek a csúcsok közötti oda-vissza bejárható kapcsolatok miatt kellenek, és így a kapcsolat két oldalán szereplő mindkét csúcsnál előfordulhatnak. Minden esetben az egyik irányban a címke nevével jelöljük az utat, a másikban a *back* módosítóval (pl. *{incl, back}*).

lexikális elemek. Az *elex, clec, tlex, flex* éleken a különböző programkonstrukciókat alkotó lexikális elemek csúcsaihoz juthatunk el. A lexikális elemek részeit a *stop, start, next, last* éleken haladva kapjuk. A tokenek közül az elsőt a *start*, a következőt a *next* és az utolsót a *last* címke adja. A lexikális elemekhez is tartoznak különböző attribútumok, de ezek a bonyolultság méréséhez szintén nem fontosak.

2.1.5. Az útvonal kifejezések szintaxisa

A mértékek lekérdezése tehát a forrásszöveg alapján felépített, és szemantikus információkkal kiegészített $\mathcal{SG}$ gráfon definiált P útvonalak bejárásán keresztül (lásd: 2.1 szintaxis leírás [23, 22]), majd a gyűjtött információk felhasználásával (darabszám, útvonalhosszak, mélység, stb.) történik.

Útvonal kifejezések

```
path()    = [PathElem]

PathElem = Tag | {Tag, Index} | {Tag, Filter}

Tag       = atom() | {atom(), back}

Index     = integer()
          | {integer(), integer()}
          | {integer(), last}

Filter    = {Filter, 'and', Filter}
          | {Filter, 'or', Filter}
          | {'not', Filter} | {Attrib, Op, term()}

Attrib    = atom()

Op        = '==' | '/=' | '=<' | '>=' | '<' | '>'
```

2.1. ábra. Az útvonal kifejezések szintaxisa [23]

Az útvonal kifejezés a szemantikus gráf éltípusainak listája, amely lista a start csúcsból induló és egy tetszőleges csúcsba, vagy csúcsokba mutató utakat ír le.

A *path()*, vagyis az útvonal kifejezés élcímkék szekvenciáját definiálja. A *PathElem* egy csúcsba vezető címkéből és az élen haladás irányát megadó atomból (*{Tag}*) áll (pl.: *{atom(), 'back'}*), vagy egy ugyanilyen címke és irány párból és egy szűrőből (*{Tag, Filter}*, vagy (*{Tag, Index}*)).

Az *Index* az azonos címkéjű élek sorszámát írja le, és a *Filter* specializálja az útvonalat, pl.: *{func, {{name '==' 'f'}, 'and', {arity '==', 1}}}*.

Egy és több útvonal kifejezés esetén is listában kell szerepeltetni az útvonalakat (*[PathElem]*). Példaként, ha meg szeretnénk találni az összes modult a gráfban, készítenünk kell egy olyan útvonal kifejezést, amely a start csúcsból kiindulva a *module* címkéjű éleken eljut az összes modul típusú csúcshoz. Ezután, ha meg szeretnénk keresni a függvény típusú csúcsokat a *func* címkéjű éleken kell haladnunk. Ezt az útvonalat egy útvonal kifejezésként is definiálhatjuk: *{[module, func]}*.

Az útvonal kifejezések, valamint a gráf bejárást végző algoritmus segítségével az útvonalak végén található gráf csúcsokhoz lekérdezhető minden az adott csomópontot jellemző attribútum, így ezzel az információval is számolhatunk a mértékek kiszámítása során. A bonyolultsági mértékek kiszámításánál a modul és a függvény programelemeket reprezentáló gráf csúcsok az érdekesek.

Egyszerűbb mértékek kiszámításához elég az útvonalakon haladva kapott csúcsok listájának a hosszát meghatározni, de bonyolultabb mértékek is kiszámolhatók a kapott listaelemek attribútumainak a vizsgálatával. Az $\mathcal{SG}$ gráf elemző algoritmusa az eredeti $\mathcal{G}$ gráf elemzőjével ellentétben a bejárás során az érintett modul és függvény típusú csomópontokat átmenetileg tárolja, lehetőséget biztosítva ezzel a kapott eredmények további lekérdezésekkel és szűrőkkel való finomítására, és így a több szintű, a megelőző elemzési folyamatok eredményétől függő lekérdezések futtatására.

2.1.6. Bonyolultság mérése útvonal kifejezésekkel

Az útvonal kifejezések vizsgálatához elemezzük a 2.2 forrásszöveget, amely két rövid függvény leírását tartalmazza. Az első függvény, a *fun1/2* két ággal rendelkezik, amely ágak (*clause*) összegzik egy tetszőleges, paraméterként kapott lista elemeit egy akkumulátorként használt változóban.

A lista bejárását követően a második ág adja az összegzés eredményét. A *fun1/1* a megfelelő paramétereket előállítva hívja meg a *sum/2* függvényt. Amennyiben meg szeretnénk határozni a példaprogram szövegét jellemző bonyolultsági mértékeket, elsőként a programszöveg alapján elő kell állítanunk a szintaxisfát, majd abból az $\mathcal{SG}$ gráfot

Erlang modul

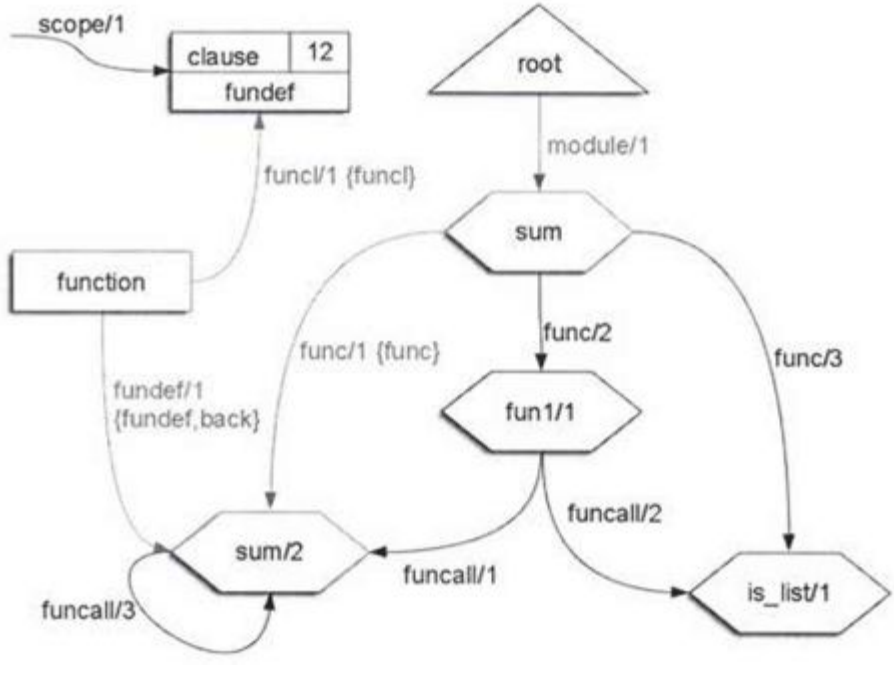
```
-module(exampmod).
-export([fun1/1]).

sum(Acc, [H|T]) ->
      sum(Acc + H, T);
sum(Acc, []) ->
      Acc.

fun1(M)->
   case M of
        {A, B} -> sum(0, [A, B]);
        List when is_list(List) ->
                              sum(0, List);
        _ ->  0
   end.
```

2.2. ábra. Egyszerű Erlang modul

2.11. megjegyzés. *A 2.2 programon alkalmazott elemzés célja jelen esetben az, hogy meghatározzuk a sum/2 függvény ágainak a számát, mivel ez a feladat megfelelően egyszerű ahhoz, hogy az eredményt fejben is követni tudjuk a 2.3. ábra alapján.*



2.3. ábra. Csomópontok szemantikus gráfban

Az $\mathcal{SG}$ felépítését követően a számítási folyamat következő lépése, hogy megkeressük a függvény modulját reprezentáló gráfcsúcsot, a megfelelő útvonal kifejezés elkészítésével. A *root*, vagyis gyökér csúcsból indulva a *module* címkéjű éleket kell elérnünk, majd tovább haladva a *func* címkéjű éleken megtalálnunk a modulban szereplő függvényeket.

Ha a függvényeket reprezentáló csúcsok rendelkezésre állnak, szűrnünk kell az őket tartalmazó listát a *sum/2* függvényre, annak nevével és az *aritásával* kiegészített útvonal kifejezés használatával, amelyet a következő formában adhatunk meg: $[\{func, \{\{name,'=','sum'\}, \{arity,'=',2\}\}\}\}]$.

2.12. megjegyzés. *Az útvonal leírásához a 2.1. listában közölt szintaxist használtuk.*

Az úton végighaladva, a keresett függvény csomópontját kapjuk amennyiben az létezik, majd ebből a csúcsból a *[{fundef, back}]* éleken érhetjük el a függvény definícióját és a *{funcl}* éllel befejezve az utat a függvény ágait (*clause*) reprezentáló gráf csúcsokat.

2.13. megjegyzés. *(Az első függvény ágat a függvény definíciójaként tartjuk nyilván, de a bonyolultság mértékek kiszámítása során minden ágat azonosnak kezelünk). A teljes útvonal kifejezést a 2.4. programlistában találjuk meg.*

Az így definiált útvonal kiértékelése során egy listát kapunk eredményül. Ebben az esetben, vagyis a függvény ágak kiszámításához elég a lista hosszát meghatározni, hogy megkapjuk a függvény ágainak a számát. (Az eredmény a *number_of_funclauses* mérték, amely formális és informális definícióját megtaláljuk a 2.1.7. fejezetben)

Útvonal kifejezések

```
[module] ...

[{func, {{name, '=', 'sum'}, {arity, '=', 1}}}] ...

[{fundef, back}, funcl]) ...
```

2.4. ábra. Alacsony szintű útvonal kifejezések függvény ágainak megszámlálására

Összefoglalva az eddig bemutatottakat, a komplexitási mértékek bármelyikét megkapjuk, ha definiáljuk az $\mathcal{SG}$ gráfon értelmezett és az adott mérték kiszámításához szükséges P útvonal kifejezést, majd elemezzük az eredményt. Ez a számítási folyamata az elemző algoritmus futása során három jól elkülöníthető fázisra bontható.

1. A forráskód elemzésével felépítjük a szintaxis fát, majd kiegészítjük azt az összes statikus elemzéssel nyerhető szemantikus információval. (Amennyiben az $\mathcal{SG}$ gráf már rendelkezésre áll, a folyamat két lépésből áll.)

2. Az így létrejött gráfon minden mérni kívánt bonyolultsági mértékhez megkonstruáljuk, majd kiértékeljük a megfelelő útvonal kifejezéseket.

3. A kapott csomópont lista alapján kiszámítjuk a kívánt bonyolultsági mértéket a listában szereplő csomópontok, és azok attribútum értékeik alapján.

2.14. megjegyzés. *A lista elemei alapján legtöbb esetben közvetlenül kiszámolhatóak a jellemző bonyolultsági mérték, de egyes esetekben további elemzésekre is szükség lehet. Vannak olyan mértékek, amelyek több más érték összegéből, vagy különbségéből, néha azok bonyolultabb kombinációjából állnak elő.*

2.1.7. Összefoglalás

Az fejezetben ismertettük az Erlang programok forrásszövegének tárolására, valamint azok bonyolultságának mérésére kifejlesztett adatstruktúrát, amelyet szemantikus gráfnak neveztünk el.

A szoftver bonyolultság méréséhez a forrásszöveg elemzése során létrehozott, a forráskódot jellemző szemantikus információkat leíró, adatfolyam, és hívási gráffal kiegészített RefactorErl $\mathcal{G}$ gráfját vettük kiindulási alapnak és definiáltuk a bonyolultsági mértékek tárolására és mérésére alkalmas adatstruktúrát, az $\mathcal{SG}$ gráfot.

A szemantikus gráf, és formális definíciójának bemutatása mellett megismerhettük azt a gráf bejárására alkalmas alacsony szintű nyelvet, amely segítségével a bonyolultsági mértékek méréséhez szükséges útvonal kifejezéseket írhatunk. Az útvonal kifejezések a későbbiekben segítségünkre lesznek a különböző strukturális bonyolultsági mértékek kiszámításában, valamint a strukturális bonyolultságot lekérdező magas szintű nyelv megalkotásában (lásd: 2.2.5. fejezetben), amely nyelv a szemantikus gráfhoz képest magasabb absztrakciós szinten teszi lehetővé az Erlang programokat jellemző bonyolultsági mértékek lekérdezését. Az útvonal kifejezéseket, és a hozzájuk megkonstruált elemzőt használja a bonyolultsági mértékek alapján az automatikus program transzformációk végrehajtására kifejlesztett, és a 4. fejezetben bemutatott elemző és transzformáló algoritmus is a 3. fejezetben ismertetett transzformációk végrehajtásához szükséges mérések elvégzésére.

A bonyolultságot elemző algoritmus képes a bonyolultsági mértékek kiszámított értékeit tárolni a szemantikus gráf $\mathcal{SG}$ modul, valamint függvény típusú csomópontjaihoz, hogy a bonyolultabb elemzések elvégzése minél kevesebb erőforrás ráfordítással járjon.

A tárolás elméletben egy, a szemantikus gráfhoz hasonló, és azzal konzisztens gráfban tárolható, amely a modul és a függvény csomópontokhoz nyilvántartja a rajtuk mért értékeket, de a bonyolultságot leíró információ ugyanabban a tárban helyezkedik el, amely az eredeti gráfot is tartalmazza. Erre azért van szükség, hogy a frissítések során a konzisztencia ellenőrzése és megtartása (minden transzformációt követően) egyszerűbb és gyorsabb legyen.

A mértékek ilyen módon történő tárolása hatékony módszer, mert a forráskód megváltoztatása, (a szemantikus gráf $\mathcal{SG}$ transzformációja) során mindig nyomon lehet követni, hogy mely gráf csomópontok, vagyis mely programelemek változtak meg. A megváltozott programrészeknél lehetőség nyílik a bonyolultsági mértékek automatikus újraszámolására (részletesen lásd: 3. fejezetben).

Ez a tulajdonság számos lehetőséget nyújt a számunkra. Ha az elemzett forráskódot átalakítottuk és a bonyolultság esetleg rossz irányba változott, ezt a változást jelezni lehet, és az elemző ezen tulajdonsága magában hordozza az automatikus programtranszformációk lehetőségét is. A problémával szintén a dolgozat 3. és 4. fejezetében részletes foglalkozunk.

2.2. Erlang programok bonyolultsági mérőszámai

2.2.1. Bevezetés

A fejezet azokat a szoftver bonyolultsági mértékeket mutatja be, amelyeket a programtranszformációk hatásainak a mérésére használunk fel a későbbiekben.

A mértékek összeállításával nem az a célunk, hogy általános felhasználású szoftver bonyolultsági mértékeket mutassunk be, hanem az, hogy az Erlang nyelvet jellemző, összetett mértékrendszer segítségével lehetővé tegyük a programok bonyolultságának a mérését. A céljaink közé tartozik az is, hogy kimutathatóvá tegyük a programokban található konstrukciós hibákat, valamint biztosítsuk a lehetőségét a forráskód kezelhetetlenül bonyolult részeinek az automatikus javításának.

A mértékek bemutatását megelőzően a 2. fejezetben megismerhettük azt az adatszerkezetet, amely segítségével a mértékeket az adott programra ki tudjuk számolni. (A bemutatásra kerülő mértékek mind implementálva vannak a *RefactorErl* szoftverben.)

2.2.2. A fejezetben ismertetésre kerülő eredmények

A fejezetben bemutatjuk az általunk létrehozott összetett mértékrendszert, amely alkalmazásával viszonylag nagy biztonsággal jellemezhető az Erlang programok forrásszövegének strukturális bonyolultsága. Az általunk használt bonyolultsági mértékek mindegyikét definiáljuk és felhasználjuk őket a dolgozatban bemutatásra kerülő további tézisek megalkotásához. A bonyolultsági mértékek mindegyikét ellenőriztük, hogy azok megfelelően mérik-e a forrásszöveg bonyolultságát. A mérések során kapott eredményeket és azok elemzését a 6. fejezetben közöljük.

2.2.3. Az Erlang nyelv mérésére kidolgozott mértékrendszer

A 2.2 táblázatban szereplő és a fejezetben ismertetett tézisek alapjául szolgáló bonyolultsági mértékek mindegyike az Erlang programok releváns tulajdonságait méri. (A mértékek között vannak olyanok, amelyeket az imperatív, vagy objektum orientált paradigmához használt mértékek közül vettünk át és kisebb átalakításokkal sikeresen alkalmazhatunk funkcionális programok mérésére.)

A mértékek kidolgozása és összeállítása során arra törekedtünk, hogy az Erlang nyelvben használható általános, vagy speciális nyelvi elemeket és konstrukciókat mérhetővé tegyünk és a forrásszöveg struktúráját és bonyolultságát viszonylag nagy pontossággal tudjuk jellemezni.

A másik szempont a mértékek összeállítása során az volt, hogy a forráskód minőségét javító transzformációs lépések (lásd: 4. fejezetben) hatása mérhetővé váljon, valamint az automatikus programtranszformációk nagy része a bonyolultsági mértékek figyelembe vételével, és a mért eredményekre alapozva elvégezhető legyen.

Az ismertetésre kerülő bonyolultsági mértékek nem csak külön-külön alkalmazhatóak a forráskódra, hanem több mérték kombinációjából újabb mértékeket tudunk előállítani.

Az összetett mértékek készítésére az elemző és bonyolultságot mérő algoritmus lehetőséget nyújt egy speciális, Erlang nyelvű *interface*-en keresztül, és az ahhoz képest magasabb absztrakciós szinten elhelyezkedő lekérdező nyelv használatával, amely a 2.2.5. fejezetben részletes bemutatásra kerül.

Mérték megnevezése	**Mérhető program-konstrukciók**
module_sum	module
line_of_code	module/function
char_of_code	module/function
number_of_fun	module
number_of_macros	module
number_of_records	module
included_files	module
imported_modules	module
internal_cohesion	module
function_calls_in	module
function_calls_out	module
cohesion	module
function_sum	module/function
otp_used	module
max_depth_of_calling	function/module
max_depth_of_cases	function/module
max_depth_of_structs	function/module
number_of_funclauses	function/module
number_of_exceptions	function/module
branches_of_recursion	function/module
mcCabe	module/function
calls_for_function	function
calls_from_function	function
number_of_funexpr	function/module
number_of_messpass	function/module
fun_return_points	function/module
average_size	module/function
average_length_of_line	module/function
max_length_of_line	module/function
extarnal_calls	function
internal_calls	function
extarnal_calls_from	function
internal_calls_from	function
number_of_variables	function/module
length_of_name	function/module

2.2. táblázat. Implementált szoftver bonyolultsági mértékek

A bonyolultsági mértékek mindegyikét, még az ismertebbeket is pontosan definiáljuk, mivel azok eltérhetnek az azonos nevű, imperatív, vagy objektum orientált nyelvek mérésére használtaktól.

A mértékeket egy speciális adatszerkezet, a szemantikus gráf $\mathcal{SG}$ (lásd: 2. fejezetben) elemzése során gyűjtött információk felhasználásával számítjuk ki.

A definíciók megértéséhez, vagy kiegészítéséhez sok esetben meg kell adnunk az adott mérték kiszámításához szükséges gráf bejáró útvonalakat. Mindezek miatt be kell vezetnünk, egy nem túl bonyolult jelölésrendszert, amely segít a gráfon értelmezett utak leírásában.

- A bonyolultságot elemző algoritmusban a $\mathcal{SG}$ szemantikus gráf bejárása az $n \in N$ csomópontokat összekötő élek címkéinek megadásával történik. Minden csomópontnak van típusa $n^{(type)}$ és minden típusba más, egyszerre többféle címkéjű él is vezethet be és vezethet ki onnan. Pl.: az $n_i \xrightarrow{*} n_j$, jelentése, hogy $n_i \in N$ csomópontból $* \in El$ él halad a n_j csomópontba.

- A szemantikus gráf csomópontjainak típusát az $n^{(type)}$, több azonos típusú csomópont esetén a $n_i^{(type)}$ formában adjuk meg. A csomópontok típusa lehet pl.: $root, mod, func, form, expr, clause, token, ...stb.$ Az elnevezéseket minden mértéknél közöljük.

- A csúcsokat összekötő élek címkéit a $module, func, fundef, expr, stb.$ nevekkel jelöljük. A neveket kiegészíthetjük módosítókkal, mint a *back*, amely például az élen visszafelé haladást jelöli. A címkéket kiegészíthetjük szűrő feltételekkel, amelyek a nevet specializálják azokat, mint a $\{type = macro\}$.

- Az útvonalakat alkotó éleket a $\xrightarrow{ctype[1..*]}$ formulával adjuk meg, ahol a *ctype* az adott él típusát jelöli, vagyis azt, hogy milyen csomópont típusba tart. A [1..*] jelentése, hogy az adott él a csomópontból kiindulva legalább egy, de akár több azonos csomópontba is tarthat (pl.: több azonos címkéjű él).

- Néhány mérték kiszámításnál, ahol az útvonalat leíró élcímkék hasonló, vagy közel azonos típusú csúcsokba tartanak és az élek címkéje nem lényeges, az élcímkéket összefoglaló névvel adjuk meg. Ilyen eset, mikor például egy kifejezés (*expr*) beágyazott rész kifejezéseit keressük a $clause, visib, sub, ...$ útvonalhoz hasonló útvonalakon haladva. Ekkor a hosszú címkesorozat helyett az egyszerűség kedvéért egy általánosított nevet (mint a *sub_expr*) fogunk használni (a mérték megértése szempontjából az általánosított címkesorozat elemei nem relevánsak, mivel a sokféle címke, minden esetben egy adott típusba tartozó csúcshoz vezet).

- Amennyiben egy címkét a $\xrightarrow{*[1..*]}$, vagy $\xrightarrow{*}$ formával adunk meg, azt írjuk le, hogy a kiindulási csomópontból bármilyen típusú élcímkén továbbhaladhatunk.

- A $\overset{*\wedge ctype}{\longrightarrow}$ formulával (pl.: $\overset{*\wedge case\_expr}{\longrightarrow}$) azt definiálhatjuk, hogy bármely élen haladva az útvonalnak egy adott típusú éllel (az élen elérhető csúcsban) kell végződnie.

- A gráf útvonalak bejárásának eredménye legtöbb esetben lista, amely lehet üres, $(L = [], |L| = 0)$, vagy több elemű $L = [l_1, ..., l_n], n = |L|$, ahol az elemek az SG szemantikus gráf programelemeket reprezentáló csúcsai. (Az adott útvonalon kiszámítható mérték eredménye 0, ha a lista üres.)

- Előfordulhat olyan eset, hogy összetett útvonalakon keresztül juthatunk el egy adott csomópontokhoz, de az útvonalak bejárása során érintett köztes csomópontokra nincs szükségünk (nem kell gyűjtenünk ezeket).
  Ez esetben a $seq(path_1, path_2, ..., path_n)$ jelölést fogjuk alkalmazni, amely az útvonal eredménylistáját szűri ezen szempont szerint.

- Az $e_1|...|e_n \wedge e$ jelölésű élsorozat egy utat definiál oly módon, hogy a | jelekkel elválasztott címkék mindegyike szerepelhet az útvonalon, de a záró él e típusú kell legyen.

- A $*ctype$ jelentése, hogy minden változatát vesszük az *megadott* típusú éleknek Pl.: $(if\_expr, case\_expr, stb.)$ helyett használhatjuk a $*\_expr$ címkét.

- Egyes lekérdezések listák listáját adják eredményül, mint amikor részgráfok mélységét, gráf útvonalak hosszát, a beágyazottságot, vagy a hívási mélységet vizsgáljuk.

  Az ilyen típusú eredményeket a $par(e_1, e_2, ..., e_n)$ függvény állítja elő. Az eredménylista az adott (mért) csomópontból kiinduló és az $e_1, e_2, ..., e_n$ élek által adott útvonal(ak) párhuzamos bejárásával kapott listákat tartalmazza. (A párhuzamosság ebben az esetben azt jelenti, hogy minden utat külön be kell járnunk.)

- Az gráfon értelmezett utakra alkalmazhatjuk a $count(e)$ függvényt, amely megadja, hogy az útvonalon hány darab e típusú él szerepel.

  Példaként a $\overset{sub\_expr\wedge count(case\_expr)}{\longrightarrow}$ útvonal a kiindulási csúcsból minden *expr* típusú élen keresztül haladva megszámolja a *case_expr* típusú kifejezéseket. Az ilyen útvonal kifejezések nem listákat, hanem numerikus értéket adnak eredményül.

- Az utakra alkalmazhatjuk a $collect(e)$ függvényt, amely összegyűjti az útvonalon szereplő e típusú éleket.

- A mértékek tárgyalásánál a 2.1 táblázatban bemutatott élcímkéket és a definíciókban a mértékek neve helyett azok kezdőbetűiből alkotott rövidítéseket használjuk, pl.: $LOC = lines\_of\_code$, vagy $MS = module\_sum$.

Minden lekérdezés, vagyis útvonal a gráfban kizárólag az $n^{(root)} \in N$ típusú csomópontból indul (ez az elemző algoritmus által használt AST tulajdonságaiból adódik [24]). Számos esetben a modulokra vonatkozó mérték kiszámításához elsőként az alábbi útvonalat kell bejárni, majd ebből kiindulva specializálni az eredményt az útvonal kifejezés kibővítésével:

$$n^{(root)} \xrightarrow{\textbf{module}[1..*]} [n_1^{(mod)}, \ldots, n_l^{(mod)}].$$

A függvény csomópontok mértékei esetén a

$$n^{(root)} \xrightarrow{\textbf{module}[1..*]} [n_1^{(mod)}, \ldots, n_l^{(mod)}] \xrightarrow{\textbf{func}[1..*]} [n_1^{(func)}, \ldots, n_l^{(func)}]$$

gráfútvonalból kell kiindulnunk. A bejárás eredménye mindkét útvonal esetén lista, amely a keresett típusú csomópontokat tartalmazza.

A definíciókban a $\mathcal{SG}$ szemantikus gráf összes modulját az M, az összes függvényt az $F(M)$ formulával jelöljük, ahol az $M = \{m_1, ..., m_k\}$ Erlang modulok.

Az $F(m_i)$ a m_i modul által tartalmazott függvényeket jelöli (modulban definiált függvények).

2.2.4. Az alkalmazott bonyolultsági mértékek

Értékes sorok száma. Az *effective_lines_of_code* a kódrészlet, konkrétabban a függvény, vagy modul azon sorainak számát adja, amelyek forrásszöveget tartalmaznak. Az *effective* jelző tehát azt jelenti, hogy az eredményben nem szerepelnek sem az üres, sem a kommenteket tartalmazó sorok.

2.15. definíció. [*effective_lines_of_code*] *Jelölje a forrásszöveg összes sorát $L(M)$ és a csak kommenteket tartalmazó sorokat $K(M)$ ($K(M) \subset L(M)$). Ekkor a forrásszöveg értékes sorainak a száma, vagyis az:*

$$ELOC(M) := |L(M) \setminus K(M)|\square.$$

A mérték alkalmazható az $\mathcal{SG}$ gráf tetszőleges moduljára: $ELOC(m_i) := |L(m_i) \setminus K(m_i)|$ és használhatjuk modulok egy csoportjának mérésére is:

$$ELOC(m_1, ..., m_k) := \sum_{i=1}^{k} ELOC(m_i).$$

Mindezek mellett az *ELOC* mértéket alkalmazhatjuk függvények egy csoportjára $ELOC(f_1, ..., f_k)$ is. A mért függvények több modulból is kikerülhetnek. Az egyetlen kikötés, hogy az $\mathcal{SG}$ szemantikus gráf tartalmazza azokat. Fontos megjegyezni, hogy a $ELOC(m_i) \neq ELOC(F(m_i))$, vagyis a modulra mért érték nem azonos a függvényeire mért eredmények összegével, mivel a modul a függvények lexikális elemein kívül tartalmazhat rekordokat, makrókat és egyéb elemeket.

Amennyiben a szemantikus gráf felől közelítjük meg a mérték kiszámítását, a következő gráfútvonalat kell definiálnunk:

$$n^{(root)} \xrightarrow{\mathbf{seq}(*\wedge *\mathbf{lex})} [n_1^{(token)}, \ldots, n_l^{(token)}]$$

A $(token)$ a *token* típusú csomópontokat jelöli, a (lex) a gráf csomópontokat a lexikális elemekkel összekötő élcímke. Jelen esetben a $*$ bármely él címkéje lehet, kivéve a lexikális elemekhez vezető éleket $(elex, clex, flex)$, mert azok kizárólag az útvonal végén szerepelnek.

2.16. állítás. *A fenti gráfútvonal bejárása során kapott eredmény listát jelölje L, és az L egy p eleméhez kötött sorvége jelek halmazát jelölje $p\downarrow$. Ekkor a mérték eredménye, vagyis* $ELOC = |\{p \mid p \in L \wedge |p\downarrow| > 0\}|$

2.17. magyarázat. *A mérték kiszámításához tehát minden olyan élet be kell járni a gráfban, amelyek a forrásszöveg lexikális elemeihez vezetnek. Az éleken keresztül megtalálható csomópontokhoz a különböző gráf útvonalak végén a $*lex$ típusú éleken keresztül lehet eljutni. Az L eredménylistában meg kell keresni azokat a lexikális csomópontokat, amelyekhez az elemző algoritmus hozzákötötte a sorvége jelet, majd ezeket kell összeszámolni (ahol több sorvége jel található, ott is egyet kell az eredményhez adni, mivel az üres sorokban szereplő sorvége jel az előző sor utolsó lexikális eleméhez van kötve).*

Átlagos sor hossz. Az *average_length_of_line* mérték az adott függvényben, vagy modulban található sorok hosszának az átlagát adja vissza. (A strukturált bonyolultság lekérdezésére kifejlesztett (és a 2.2.5. fejezetben bemutatott) nyelvtan segítségével, a *max_length_of_line* mértékhez hasonlóan ez az érték is előállítható, ha alkalmazzuk annak *avg*, vagy *sum* nevű filtereit, de a belső *interface* számára fontos, hogy a mértékek a lekérdező nyelvtől függetlenül is elérhetőek legyenek.)

Ez a mérték annak ellenére, hogy nem tűnik fontosnak, nem kihagyható, mivel az egyes programozók és cégek programjaiknál előírják az átlagos sorhosszok alkalmazását, valamint a forrásszöveg olvashatóságát rossz irányba befolyásolhatják a túl hosszú sorok.

Az eredmény kiszámításához a *ELOC* mértéket kell alapul vennünk és a 2.1.7 fejezet 2.15. definícióban leírtak szerint ki kell számítanunk a sorok hosszát, majd a kapott eredményt átlagolnunk, vagyis az

$$ALOL(M) = \lfloor \frac{ELOC(M)}{CHOC(M)} \rfloor.$$

Az *ALOL* mérték nem veszi figyelembe az eredmény meghatározása során az üres, vagy a kizárólag kommenteket tartalmazó sorok hosszát. A szemantikus gráf bejárása szintén azonos az *ELOC* mértéknél bemutatottal. A mértéket alkalmazhatjuk függvények, vagy modulok mérésére is, de fontos megjegyezni, hogy az

$$ALOL(m_i) \neq ALOL(F(m_i))$$

vagyis a függvények átlagos sorhossza nem azonos a modul átlagos sorhosszával.

Karakterek száma. A *chars_of_code* a programszöveg karaktereinek számát adja vissza. A mérés során kapott érték nem tartalmazza a kommentek jeleinek a számát, valamint nincs benne a sortörés és egyéb *white space* karakterek száma sem.

2.18. definíció. *[chars_of_code] Ha $L(M)$ a forrásszöveg összes sora és $K(M)$ a csak kommenteket tartalmazó sorok, $S = L(M) \setminus K(M)$ az értékes sorok halmaza, $s_i \in S$ ennek a halmaznak egy eleme, w_i az s_i sorban található kommentek, és white space karaktereinek halmaza, és $n = |S|$, akkor*

$$CHOC(M) = \sum_{i=1}^{n} (|s_i| - |w_i|) \quad \square$$

A mérték az *effective_lines_of_code*-hoz hasonlósan alkalmazható függvények csoportjára $CHOC(f_1, ..., f_j)$, de használhatjuk egy modul $COC(m_i)$, vagy modulok egy csoportjának $COC(m_1, ..., m_j)$ mérésére. Mivel a *CHOC* mérték alapja a sorok és a sorok jeleinek a száma, itt is igaz, hogy a

$$CHOC(m_i) \neq CHOC(F(m_i)),$$

vagyis a modulra mért összesített érték nem azonos a függvényeire mért eredmények összességével. A mérték kiszámításához szükséges gráf útvonal azonos a *effective_lines_of_code* mértéknél definiálttal, de itt az útvonalak bejárásával kapott csomópontokban talált lexikális elemekben található jelek számát kell összegezni ahhoz, hogy az eredményét megkapjuk.

2.19. állítás. *Jelölje a megfelelő gráfútvonalon kapott eredmény listát L, jelölje az L egy p eleméhez kötött kommentek jeleinek számát $c(p)$, valamint az ugyanezen elemekhez kötött akárhány sorvége jelek számát $db(p\downarrow)$, és a $p \in A$ által reprezentált forrásszöveg elemeinek hosszát $length(p)$. Ekkor a mérték eredménye, vagyis*

$$CHOC(M) = \sum_{p \in L}(length(p) - c(p) - db(p\downarrow)).$$

Függvények száma. A *number_of_functions* mérték a modulokban definiált függvények számát adja vissza. Ez a mérték különösen releváns funkcionális programok jellemzése során, mivel azok nagy számban tartalmaznak függvény konstrukciókat, így a *lines_of_code* mellett, annak használatával következtethetünk a modulok méretére.

2.20. megjegyzés. *Az -import(m,f). formulával láthatóvá tett függvényeket, valamint az $F(M)$ függvényei által hívott függvények számát a mérték nem tartalmazza,*

a_i:
$\mathsf{m} : \mathsf{f}(e_1, ..., e_n)$ vagy $\mathsf{f}(e_1, ..., e_n)$

valamint a több ággal (clause) rendelkező függvények ágait nem számolja bele az összesített értékbe. Az $m : f(e)$ formula a minősített hívásokat jelöli, $f(e)$ pedig a más modulokból importált, valamint a mért modulban hívott függvényeket. Az e és $e_i \in E$ tetszőleges Erlang kifejezések, amelyek a meghívott függvény aktuális paraméterei.

2.21. megjegyzés. *Az Erlang függvények definícióját általános az alábbi formulával írhatjuk le:*

f_0:

$\mathsf{f}_1^c(p_1)$ *when* g_1 $\rightarrow$
$e_1^1, \ldots, e_{l_1}^1;$
$\vdots$
$\mathsf{f}_n^c(p_n)$ *when* g_n $\rightarrow$
$e_1^n, \ldots, e_{l_n}^n.$

Ahol f_i^c *az* i^{th} *függvény ág,* $e_i \in E$ *Erlang kifejezések* $g_i \in G$ *a függvény ágaihoz tartozó őr feltételek és* $p_i \in P$ *a függvények formális paraméter listáját alkotó minták.*

2.22. definíció. [*number_of_functions*] *A mérték eredménye az* $\mathcal{SG}$ *szemantikus gráfban szereplő összes modulra* $NOF(M) = |F(M)|$.

A mérték segítségével mérhetünk egy modult $NOF(m_i) = |F(m_i)|$, vagy modulok egy csoportját

$$NOF(m_1, ..., m_k) = \sum_{i=1}^{k} |F(m_i)|.$$

A függvény definíciókat reprezentáló csomópontokat az $\mathcal{SG}$ szemantikus gráf *mod* és *func* címkéjű élein keresztül érhetjük el (ez az él az útvonalon a modulban definiált függvények csomópontjaihoz vezet).

$$n^{(root)} \xrightarrow{\textbf{module}[1..*]} [n_1^{(mod)}, ..., n_t^{(mod)}] \xrightarrow{\textbf{func}[1..*]} [n_1^{(func)}, \ldots, n_k^{(func)}]$$

Makrók száma. A *number_of_macros* mérték az adott modulban, vagy modulokban definiált makrók, pontosabban makró definíciók számát adja meg.

2.23. megjegyzés. *Makró definíciók:*

$\mathsf{def}_1(v_1, e_1).$
$\vdots$
$\mathsf{def}_t(v_t, e_t).,$

ahol $v_i \in V$ *makró nevek és* $e_i \in E$ *kifejezések, amelyeket a makró nevére hivatkozva érhetünk el azokban a modulokban, ahol a makró definíciója látható.*

2.24. definíció. [*number_of_macros*] *Az* $m_i \in M$ *modul által tartalmazott makró definíciókat jelölje* $D(m_i) := \{def_1, .., def_t\}$, *ahol* def_i *egy makró definíció. A* $D(m_i)$ *a*

fejléc fájlokkal láthatóvá tett makrókat nem tartalmazza. Ekkor a mérték eredménye az összes modulra

$$NOM(M) = \sum_{m \in M} |D(m)|.\square$$

A mérték használható egy modulra: $NOM(m_i) = |D(m_i)|$, valamint alkalmazható modulok egy csoportjára is

$$NOM(m_1, ..., m_k) = \sum_{j=1}^{k} |D(m_j)|.$$

A NOM mérték eredménye a makró definíciókat reprezentáló, szemantikus gráfban található csomópontok száma, amelyeket az alábbi gráf útvonalon érhetünk el:

$$n^{(root)} \xrightarrow{\mathbf{seq(module,file)}[1..*]} [n_1^{(file)}, ..., n_l^{(file)}] \xrightarrow{\mathbf{form\{type=macro\}}[1..*]} [n_1^{(mac)}, \ldots, n_k^{(mac)}]$$

A makró definíciókat a *module*, majd a *file* csomópontokon keresztül haladva a *form* éleken keresztül kapjuk, de ezeket az éleket specializálnunk kell a $\{type = macro\}$ szűrő feltétel alkalmazásával. (A 2.24. definíció alapján az eredményként kapott lista a modulban használt, de nem ott definiált makrók számát nem tartalmazza.)

Rekordok száma. A *number_of_records* mérték az adott modulban definiált rekordok számát adja vissza.

Az Erlang nyelvben a rekordok használatának lehetősége sok esetben olvashatóbbá, de mindenképpen alakíthatóbbá teszi a forrásszöveget, mindezek mellett a rekordokat használhatjuk a függvény paraméterek dinamikusabbá tételére is. A *rekord update* [31] mechanizmus alkalmazásával a rekordokat a függvények formális paramétereként használva, a paraméter lista aktualizálása során nem kell minden rekord mezőt szerepeltetnünk. A rekord mezők bővítésével a függvények paraméterezésének átalakítása nélkül adhatunk extra információt a függvény ágakhoz, más ágak megváltoztatása nélkül.

Mintaillesztés során *tuple* mintára illeszthetünk rekordokat, és ez fordítva is történhet azzal a megkötéssel, hogy az n-es első eleme meg kell egyezzen rekordnévvel. Mindezért a rekordok fontos szerepet kapnak a bonyolultság vizsgálata során.

2.25. megjegyzés. *A rekordok definíciója az Erlang modulok elején szerepelhet a következő formában:*

$$\mathsf{rec}_1(n_1, f_1^1[= v_1^1], ..., f_n^1[= v_n^1]).$$
$$\vdots$$
$$\mathsf{rec}_t(n_t, f_1^t[= v_1^t], ..., f_n^t[= v_n^t]).$$

Ahol $rec_i \in R$ rekord definíciók, $n_i \in N$ rekord nevek, f_i^j a rekord mezőinek a nevei, és v_i^j (opcionális) a mezők értékei.

2.26. definíció. *[number_of_records] Az $m_i \in M$ modul által tartalmazott makró definíciókat jelölje $R(m_i) := \{rec_1, .., rec_t\}$, ahol rec_i egy rekord definíció. A $R(m_i)$ a fejléc fájlokkal láthatóvá tett makró definíciókat nem tartalmazza. Ekkor a mérték eredménye az összes modulra*

$$NOR(M) = \sum_{m \in M} |R(m)|.\square$$

A mérték használható egy modulra: $NOR(m_i) = |R(m_i)|$ és alkalmazható modulok egy csoportjára is

$$NOR(m_1, ..., m_k) = \sum_{j=1}^{k} |R(m_j)|.$$

A rekord definíciókat reprezentáló szemantikus csomópontokat a szemantikus gráfban az alábbi útvonalon érhetjük el:

$$n^{(root)} \xrightarrow{\mathbf{seq(module,file,record)[1..*]}} [n_1^{(rec)}, \ldots, n_k^{(rec)}]$$

A rekordokat tartalmazó listát a *module*, *file*, *record* címkéjű éleken keresztül kapjuk.

Fejléc fájlok száma. A *number_of_headers* mérték a modulban szereplő fejléc fájlok számát adja vissza.

Az Erlang nyelvben a több modul függvényei által közösen használt adatokat, rekordokat, makrókat, vagy szélsőséges esetekben függvényeket fejléc fájlokba rendezik, majd ezeket a fájlokat az *include(filepath)* formában elérhetővé teszik modulok számára. Ezáltal az Erlang programokban fontos szerephez jutnak, mivel a modulok kapcsolatrendszerét, a hívási gráfokat és adatfolyam analízisek eredményeit befolyásolhatják.

2.27. definíció. *[number_of_headers] Jelölje $H = \{h_1, ..., h_k\}$ a fejléc fájlokat (*.hrl), és $A = M \cup H$, valamint jelölje az include(x, y) azt, ha $x \in M$ az include(filepath) for-*

mában láthatóvá teszi az $y \in A$ fájlt. Ekkor

$$In(M) = \{(x, y) \mid y \in A \ \wedge \exists x \in M \wedge include(x, y) \wedge x \neq y\}$$

esetén $NOH(M) = |In(M)|$.

2.28. magyarázat. *Az $include(x, y)$ formulában megengedjük azt az esetet, amikor egy Erlang modul egy másik Erlang modult tesz láthatóvá.*

A mérték alkalmazható modulok egy csoportjára

$$NOH(m_1, ..., m_k) = \sum_{i=1}^{k} |In(m_i)|,$$

ahol $In(m_i) = \{(m_i, y) \mid y \in A \ \wedge include(m_i, y) \wedge m_i \neq y\}$.

A mérték kiszámításához a mért modulból kiindulva meg kell találni az ahhoz tartozó, fájlt reprezentáló gráf csomópontot, majd innen az *incl* éleken továbbhaladva az *include*, vagyis a modulban láthatóvá tett fejléc fájlokat.

$$n^{(root)} \xrightarrow{\text{seq(module,file,incl)[1..*]}} [n_1^{(file)}, \ldots, n_l^{(file)}]$$

Az *incl* élek fájlokat kötnek össze. Az élek definiálta útvonalak végén szerepelhetnek kimondottan fejléc fájlok, de lehetnek Erlang modulok is. Az éleken bejárt útvonal eredményeként kapott lista hossza, vagyis l adja az eredményt. Egy fájl kétszer nem szerepelhet a listában még akkor sem, ha több modult mérünk, és akkor sem, ha a fájlt két modul is felhasználja fejléc fájlként).

Importált modulok. Az *imported_modules* mérték az adott modulba (nem rekurzívan) importált más modulok számát adja vissza. Az import modulok száma arra világíthat rá a mérések során, hogy a mérésben szereplő modulok milyen szoros, vagy éppen laza kapcsolatban állnak egymással. Ez a fajta mérés fontos lehet annak az eldöntésében, hogy egy klaszterezési eljárás során egy modul a kapcsolatrendszer alapján mely csoportoknak legyen tagja. A mérték nem tartalmazza a minősített hívások (*module:function* alakú hívások) számát.

2.29. definíció. *[imported_moduls] Az $m_i \in M$ modul által importált modulokat jelölje $I(m_i)$. Kikötjük, hogy $m_i \notin I(m_i)$, valamint a rekurzívan importált modulokat*

nem számoljuk az eredménybe. Ekkor az

$$IM(M) = \sum_{m \in M} |I(m)|.$$

A mérték alkalmazható egy modul mérésére $IM(m_i) = |I(m_i)|$, valamint modulok egy csoportjának a mérésére:

$$IM(m_1, ..., m_k) = \sum_{i=1}^{k} |I(m_i)|.$$

Az $\mathcal{SG}$ szemantikus gráfban található importált modulokat reprezentáló csomópontok (*formok*) számát az alábbi gráfútvonalon érhetjük el:

$$n^{(root)} \xrightarrow{\mathbf{seq(module,file)[1..*]}} [n_1^{(file)}, ..., n_l^{(file)}] \xrightarrow{\mathbf{form\{type=import\}[1..*]}} [n_1^{(form)}, ..., n_k^{(form)}]$$

Az import *formokat* a *module*, majd a $file$ csomópontokon át a $form$ típusú éleken keresztül kapjuk meg, de az élcímkét specializálnunk kell a $\{type = import\}$ feltétellel.

Kohézió. A *cohesion* nevű mérték a modulok közötti összes függvény útvonal számát adja eredményül, de a belső függvény kapcsolatok számát nem méri.

2.30. megjegyzés. *Az $m_i \in M$ modulokban található függvény hívásokat két esetben tekinthetjük modulok közti függvény útvonalnak:*

- *Az első eset, ha a kapcsolatban szereplő függvény moduljára igaz, hogy tartalmazza a hívó, de nem tartalmazza a hívott f_i^{app} függvény definícióját.*
- *A másik eset, ha az m_i nem tartalmazza a hívó, de tartalmazza a hívott f_i^{app} függvény definícióját.*

2.31. definíció. *[függvényhívási kapcsolat] Az $m_i \in M$ modulok tartalmazhatnak függvény definíciókat. Minden $f_i, f_j \in F(M), f_i \neq f_j$ párra keressük meg, hogy f_i és f_j között van e hívási kapcsolat. Kapcsolat alatt azt értjük, hogy f_i hívja f_j függvényt. Ezt a következőképp jelöljük: $c(f_i, f_j)$. Az összes ilyen kapcsolat halmazát jelölje*

$$C(F(M)) = \{(f_i, f_j) | f_i, f_j \in F(M) \wedge f_i \neq f_j \wedge \exists c(f_i, f_j)\}.$$

Az m_i modulból kifelé tartó függvényhívások jelölése

$$C(m_i, F(M)) = \{(f_i, f_j) | f_i \in F(m_i) \wedge f_j \in F(M) \wedge f_i \neq f_j \wedge \exists c(f_i, f_j)\}.$$

Az m_i modulba tartó összes hívás jelölése

$$C(F(M), m_i) = \{(f_i, f_j) | f_i \in F(M) \wedge f_j \in F(m_i) \wedge f_i \neq f_j \wedge \exists c(f_i, f_j)\}.$$

Az adott modul egy függvényére az $\mathcal{SG}$ szemantikus gráfból érkező összes függvényhívást jelölje

$$C(F(M), f_j) = \{(f_i, f_j) | f_i \in F(M) \wedge f_i \neq f_j \wedge \exists c(f_i, f_j)\}.$$

Adott függvényből kifelé irányuló függvényhívásokat jelölje

$$C(f_i, F(M)) = \{(f_i, f_j) | f_j \in F(m_i) \wedge f_i \neq f_j \wedge \exists c(f_i, f_j)\}.$$

Az m_i modul belső függvényhívásai

$$C(F(m_i), F(m_i)) = \{(f_i, f_j) | f_i, f_j \in F(M) \wedge f_i \neq f_j \wedge \exists c(f_i, f_j)\}.$$

2.32. megjegyzés. *Amennyiben a 2.31. definíció alapján a $c(f_i, f_j)$ és $c(f_j, f_i)$ is fennáll, vagyis a függvények kölcsönösen hívják egymást, a kapcsolatok száma kettő. Ha egy függvény többször hív egy másikat függvényt, ezek a kapcsolatok egy útvonalnak számítanak.*

2.33. definíció. [*cohesion*] *A kohéziós mérték*

$$COH(M) = C(F(M)) - \sum_{m \in M} |C(F(m), F(m))| \square.$$

A kohézió mérésénél az $\mathcal{SG}$ gráfban található összes függvénykapcsolatból kizárjuk az olyanokat, ahol a függvényhívás ugyanabból a modulból indul ki és oda is tart.

A modulok közti függvény útvonalak kereséséhez elsőként meg kell találnunk a modulokat a *module* éleken keresztül, majd a függvényeket a $func$ éleken haladva. (Egy modul mérésénél specializálni kell az útvonalat a modul nevével.)

A $func$ éleken megkapjuk a modul függvényeit, majd a $\{funcall, back\}$ éleken át a függvényekre vonatkozó hívásokat végző függvénycsomópontokat.

$$n^{(module)} \xrightarrow{\text{seq(func,(funcall,back))[1..*]}} [n_1^{(func)}, \ldots, n_k^{(func)}]$$

Minden hívást végző függvényhez meg kell keresni a modulját, az alábbi útvonal kifejezéssel, majd párokat kell alkotnunk belőlük, amely párok tartalmazzák a hívó függvényt és a modulját $(n^{(func)}, n^{(mod)})$:

$$n_i^{(func)} \xrightarrow{\text{(func,back)}} [n_1^{(mod)}, \ldots, n_l^{(mod)}]$$

A kiindulási, vagyis a mérésben szereplő modul csomópontja eleve adott. Minden hívó függvényhez meg kell állapítani, hogy annak modulja megegyezik-e a mért modullal. Amennyiben igen, akkor a függvényhívás nem számít bele a mérték eredményébe, minden más esetben igen.

Összetartó erő. A *coupling* mérték a modulban található függvények közötti hívási kapcsolatok számát adja eredményül. A belső függvény kapcsolatok számát igen, de a kívülről jövő, vagy a modulból kifelé tartó utak számát nem, vagyis csak a belső hívások számát tartalmazza.

2.34. megjegyzés. *A függvény hívásokat jelölje*

app$_i$:
$\mathsf{m}_i : \mathsf{f}(e_1, \ldots, e_n)$ vagy $\mathsf{m}_i : \mathsf{f}(e_1, \ldots, e_n)$,

ahol m_i *a mért modul,* $m_i : f(e_1, \ldots, e_n)$ *a modul függvényének minősített hívása és* $f(e_1, \ldots, e_n)$ *a függvény hívása* m_i *modulból.* $e_1, \ldots, e_n$ *Erlang kifejezések a függvény aktuális paramétereként.* n *a hívott függvény paramétereinek a száma.*

2.35. definíció. [*coupling*] *A coupling mérték*

$$CP(m_i) = |C(F(m_i), F(m_i))|_{\square}.$$

A modulokban szereplő függvény útvonalak kereséséhez hasonlóan kell eljárnunk, mint a *cohesion* mértéknél, csak itt a kapott párok listájából ki kell válogatni azokat, ahol a hívó függvényt tartalmazó modul megegyezik a mért modullal, vagyis $\forall n_i^{(func)} \in F(m_i)$.

Modulba irányuló függvényhívások száma. A *function_calls_in* mérték minden, a modul függvényeire irányuló, de külső modulból érkező függvényhívás (*application*) számát méri.

2.36. definíció. [*function_calls_in*] *Az összes, az adott modulba irányuló függvényhívások száma* $FCI(m_i) = |C(F(M), m_i)|$.

A modulban szereplő függvény útvonalak kereséséhez elsőként meg kell találnunk az $\mathcal{SG}$ gráf moduljait a *module* éleken keresztül, majd azok függvényeit a *func* éleken haladva.

$$n^{(root)} \xrightarrow{\text{seq(module,func)[1..*]}} [n_1^{(func)}, \ldots, n_k^{(func)}]$$

(Egy modul esetén egyszerűbb a keresés, mivel nem kell minden modul minden függvényét megtalálnunk.

Ekkor a $n^{(root)}$ elemből indulva egy specializált élen keresztül egy lépésben eljuthatunk a keresett modul csomópontjához, ha az élhez kapcsolt szűrőben megadjuk a keresett modul nevét $module\{name = name(m_i)\}$.)

Végül a $\{funcall, back\}$ éleken át megkapjuk a modul függvényeire vonatkozó hívásokat tartalmazó n^{func} csomópontokat.

$$n^{(func)} \xrightarrow{\{\text{funcall,back}\}[1..*]} [n_1^{(func)}, \ldots, n_t^{(func)}]$$

Az eredmény az összes, az m_i modul függvényeit hívó függvények csomópontjainak listája. A listából ki kell szűrnünk az m_i modulból jövő hívásokat tartalmazó függvényeket. A szűréshez a $\{func, back\}$ éleken haladva minden hívást végző függvényhez meg kell keresni annak modulját és (f, m) alakú párokat alkotni belőlük, ahol a párok egyik eleme a függvény, a másik a modul.

$$\{(n_i^{(func)}, n_i^{(func)} \xrightarrow{\{\text{func,back}\}} n_j^{(mod)}) | n_i^{func} \in F(M), n_j^{(mod)} \in M\}$$

A párok listájából ki kell válogatni azokat a függvényeket, ahol a párok második eleme nem egyezik meg a mért modullal.

Modulból kifelé tartó függvényhívások száma. A *function_calls_out* mérték minden, a modul függvényeiből induló, és más modulok felé irányuló függvényhívások számát adja vissza.

2.37. definíció. *Az m_i modulból kifelé irányuló függvényhívások száma,*

$$FCO(m_i) = |C(m_i, F(M))|.$$

A modulban szereplő függvény útvonalak kereséséhez elsőként meg kell találnunk a modulokat a *module* éleken keresztül, majd a függvényeket a *func* éleken haladva.

$$n^{(root)} \xrightarrow{seq(module,func,funcall)[1..*]} [n_1^{(func)}, \ldots, n_k^{(func)}]$$

Végül a *funcall* éleken át megkapjuk az összes hívást végző függvény csomópontjainak listáját. A listában meg kell keresnünk azokat az elemeket, amelyek az m_i modulban vannak definiálva. A kereséshez a $\{func, back\}$ éleken haladva minden hívást végző függvényhez meg kell keresni az általa hívott függvényt és mindkét függvény modulját, majd $(f_c, m_c, f_{app}, m_{app})$ alakú négyeseket alkotni belőlük, ahol a párok első, és második eleme a hívó függvény és modulja, a harmadik és negyedik a hívott függvény valamint annak modulja. A kapott négyesek listájából ki kell válogatni azokat, ahol a két modul nem azonos.

Függvény hívási mélység maximuma. A *max_depth_of_calling* mérték a függvények hívási útvonalainak a hossza. A működés megértéséhez segítségünkre lehet a 2.5 példa, ahol mért legnagyobb hívási mélység három.

Hívási mélység

```
...
f([A|B], Acc) ->
   Acc0 = exec(A, Acc),
   f(B, );s
f([], Acc0)->
    acc0.

exec(A, Acc)->
   io:format("~w",[A]),
   A + Acc.
...
```

2.5. ábra. Függvények hívási mélysége

2.38. definíció. *[függvényhívási lánc hossza] A $c(f_i, f_j)$ pár egyben hívási lánc, melynek hossza egy. Jelöljön egy f_i függvényből kiinduló hívási láncot $l(f_i, f_j)$, ha $\exists f_1, f_2, ...f_m$, hogy $\exists c(f_i, f_1) \wedge c(f_1, f_2) \wedge ... \wedge c(f_m, f_j)$, és $\nexists f \in F(M)$, hogy $c(f_j, f)$, vagyis f_j nem tartalmaz függvényhívást. Ekkor az $|l(f_i, f_j)| = m + 1$*

2.39. definíció. *[max_depth_of_calling] Legyen*

$$L(f_i) = \{l(f_i,\ f_k) |\ \exists f_k \in F(M), f_i \neq f_k\}$$

az f_i-ből induló hívási láncok halmaza. Ekkor a függvény hívási mélység maximuma

$$MDOC(f_i) = \max\{|l(f_i, f_k)| \mid l(f_i, f_k) \in L(f_i)\}$$

A mértéket alkalmazhatjuk egy függvényre, függvények egy csoportjának

$$MDOC(f_1, ..., f_k) = \max \bigcup_{i=1}^{k} MDOC(f_i),$$

valamint alkalmazhatjuk modulok mérésére is. A modult jellemző hívási lánc maximuma megegyezik a modul függvényeire mért hívási láncok maximumával, ahol:

$$MDOC(m_i) = MDOC(F(m_i))$$

Az eredmény az összes modulra mérve:

$$MDOC(M) = \max \bigcup_{n \in M} MDOC(m).$$

A szemantikus gráfban a függvény hívások az *application* éleken keresztül érhetőek el. Meg kell keresnünk a mért függvény ágaiból (*clauses*) kiinduló hívási láncokat, majd ezek hosszát venni és megkeresni a hosszok maximumát.

$$n^{(func)} \xrightarrow{\text{par}(\{\text{fundef,back}\},\text{funcl,application})[1..*]} [[n_1^{(func)}, \ldots, n_t^{(func)}], ...]$$

Ahol a kapott hívási láncok listája L, és $l_i \in L, l_i = [n_1^{(func)}, \ldots, n^{(func)}]$ egy, a mért függvényből kiinduló hívási láncot tartalmazza. A kapott listák közül a legnagyobb számosságú elemszáma adja az eredményt.

2.40. állítás. *A max_depth_of_calling mérték eredménye az L listában szereplő részlisták közül a leghosszabb számosságával egyenlő.*

***Case* kifejezések maximális beágyazottsága.** A *max_depth_of_cases* a függvényben, vagy a modul függvényeiben szereplő *case* vezérlő szerkezetek beágyazottságának maximuma.

Beágyazottság

```
...
case Data of
     {Pid, D} -> D ! Pid;
     _ -> case Data of
               ...
          end
end
...
```

2.6. ábra. *Case* kifejezések egymásba ágyazása

A *case* kifejezéseket mérésekor nem azok számát, hanem a beágyazottságukat kell vizsgálnunk, hasonló módon, mint a függvény hívási mélység esetén.

2.41. megjegyzés. *Az alábbi*

c_0:
case e of
p_1 [when g_1] $\rightarrow e_1^1, \ldots, e_{l_1}^1$;
$\vdots$
p_n [when g_n] $\rightarrow e_1^n, \ldots, e_{l_n}^n$
end

Erlang case kifejezés, amelyben $e, e_i \in E$ Erlang kifejezések, $p \in P$ minták $g_i \in G$ őr feltételek az ágakban. A case kifejezések ágaiban az e_j^i kifejezések tartalmazhatnak beágyazott vezérlő szerkezeteket, többek között újabb case-eket.

2.42. definíció. *[max_depth_of_cases] Jelöljük $T(f_i)$-vel az f_i függvényben található összes case kifejezés halmazát. Jelölje $t(c_1, c_2)$ azt, ha c_1 case kifejezés valamely ága tartalmazza c_2 case kifejezést és $\nexists c_3$ case kifejezés, hogy $t(c_1, c_3) \wedge t(c_3, c_2)$.*

Jelölje $t_s(c, c_x)$ azt az esetet, hogy a c case kifejezés valamely ágában valamely mélységben tartalmazza a c_x case kifejezést, vagyis $\exists\ c_1, ..., c_n$ case kifejezések, hogy

$$t(c, c_1), t(c_1, c_2), ..., t(c_{n-1}, c_n), t(c_n, c_x).$$

A $|t_s(c, c_x)|$ beágyazás mélysége ez esetben $n + 1$.

Legyen $T_0(f_i)$ azon case kifejezések halmaza, amelyeket egyetlen $T(f_i)$ halmazbeli case kifejezés sem tartalmaz (felső szintű). Ekkor az

$$MDC(f_i) = max\{|t_s(c, c_x)| \ \ |c \in T_0(f_i), c_x \in T(f_i)\}\square.$$

A mértéket alkalmazhatjuk egy függvényre, függvények egy csoportjára

$$MDC(f_1, ..., f_j) = \max \bigcup_{i=1}^{k} MDC(f_i),$$

valamint modulok mérésére is, ahol a modult jellemző beágyazottság maximuma megegyezik a függvényeire mért maximummal: $MDC(m_i) = MDC(F(m_i))$

Az eredmény az összes modulra mérve:

$$MDC(M) = \max\{MDC(m_i)|m_i \in M\}$$

Az eredmény előállításához a szemantikus gráfban meg kell találnunk a modulokat, majd a függvények csomópontjait és ágait.

$$n^{(root)} \xrightarrow{\text{seq(module,func,\{fundef,back\},funcl)[1..*]}} [n_1^{(funcl)}, \ldots, n_t^{(funcl)}]_1]$$

Ezt követően a függvényekben a felső szintű kifejezéseket, vagyis azokat, amelyek a beágyazottakat tartalmazhatják a *visib* éleken. A következő lépésben a *clause*, *visib*, *sub*, ..., éleken haladva meg kell kell keresnünk a részkifejezéseket. Az egyszerűség kedvéért a részkifejezések felé tartó éleket jelöljük a *sub_expr* éltípussal. Ahhoz, hogy a *case* kifejezéseket számolni is tudjuk, alkalmaznunk kell a *count*(∗) függvényt.

$$n^{(funcl)} \xrightarrow{\text{par(seq(visib,sub\_expr\^{}count(case\_expr)))[1..*]}} [([n_1^{expr}, \ldots n_l^{expr}], int())_1, ...]$$

A függvény minden felső szintű (*top_expression*) kifejezéseiből kiinduló összes útvonalon megszámolja a *case* kifejezéseket, majd az eredmény listában kapott párok közül kiválasztja azt, amelynek a második eleme a legnagyobb numerikus értéket tartalmazza.

2.43. állítás. *A fenti útvonal kifejezés eredményeként előálló és a szűrési feltételnek megfelelő párok listáját jelöljük L-el. Ekkor a max_depth_of_cases mérték eredménye az L lista számossága.*

Kifejezések maximális beágyazottsága. A *max_expr_depth* mérték az adott függvény függvényben szereplő vezérlőszerkezetek, vagy egyéb kifejezések (*üzenet küldő, fogadó kifejezések, függvény kifejezések, ...*) egymásba ágyazottságának mértékének maximumát méri.

A mérték kiszámítása közel azonos a *max_depth_of_cases* mértéknél látottakkal, annyi különbséggel, hogy a definícióban, valamint a kiszámításhoz szükséges útvonal kifejezésben nem csak a *case* kifejezéseket, hanem $if, try, function$ és egyéb Erlang programokban szereplő kifejezések beágyazottságát is vizsgáljuk.

2.44. definíció. *[max_expr_depth] Jelölje $t_s(k, k_x)$ azt az esetet, hogy a k kifejezés valamely mélységben tartalmazza a k_x kifejezést, vagyis $\exists\ k_1, ..., k_n$ kifejezések, hogy*

$$t(k, k_1), t(k_1, k_2), ..., t(k_{n-1}, k_n), t(k_n, k_x).$$

A $|t_s(k, k_x)|$ beágyazás mélysége ez esetben $n + 1$.

Legyen $T_0(f_i)$ azon kifejezések halmaza, amelyeket egyetlen $T(f_i)$ halmazbeli kifejezésbe sincsenek beágyazva (felső szintű kifejezések). Ekkor az

$$MED(f_i) = max\{|t_s(k, k_x)| \ \ |k \in T_0(f_i), k_x \in T(f_i)\}\square.$$

A mértéket alkalmazhatjuk egy függvény $MED(f)$, függvények egy csoportjának $MED(f_1, ..., f_j)$, valamint modulok mérésére is, ahol a modult jellemző beágyazottság maximuma megegyezik a függvényeire mért maximummal:

$$MED(m_i) = MED(F(m_i))$$

Az eredmény az összes modulra mérve:

$$MED(M) = \max\{MED(m_i)|m_i \in M\}$$

2.45. állítás. *Mivel ez a mérték nagyon hasonlít a max_depth_of_cases-re, de nem csak egy speciális kifejezést mér, hanem többféle típust is, vagyis a count(∗) függvényt*

nem alkalmazzuk az útvonalon, hanem az útvonal kifejezés eredményeként kapott, beágyazott kifejezések listájának a hosszát vesszük alapul.

$$n^{(funcl)} \xrightarrow{\text{par(seq(visib,sub\_expr))[1..*]}} [[n_1^{expr}, \ldots n_l^{expr}]_1, \ldots]$$

A függvény minden legfelső szintű (top_expression) kifejezéseiből kiinduló, egymásba ágyazott kifejezéseken keresztül haladó útvonalat be kell járnunk, majd az eredmény listában kapott $[n_1^{expr}, ..., n_n^{expr}]$ alakú listák közül ki kell választani a leghosszabbat, amely darabszáma a mérték eredményét adja.

Függvény ágak száma. A *number_of_funclauses* mérték a függvény *clause*-ainak (ágainak) a számát adja. A függvény definícióját, és minden ágat az eredményhez számolunk. Az azonos nevű, de más aritású függvényeket nem adjuk az összeghez (az erlang nyelvű programokban szerepelhetnek azonos nevű függvények egy modulban, csak az aritásuknak (*formális paraméterek száma*) kell különbözniük. Ekkor a két azonos nevű függvény teljesen független egymástól). Mindezek alapján a 2.7 példában a *number_of_funclauses* értéke kettő, mivel az *f* függvény egy és két paraméteres formában két különböző függvénynek számít. Az Erlang programokban a *case*, valamint az *if* vezérlő szerkezetek helyett a több ággal rendelkező, (overload) függvények alkalmazása megszokott módszer.

Azonos nevű függvények

```
-export([f/2, f/1]).

f(Fun, [H|Tail])->
  Fun(H),
  f(Tail);

f(_, [])->
  ok.

f(A)->
 A + 10.
...
```

2.7. ábra. Függvény klózok száma

Az Erlang függvények (hasonlóan az objektum orientált nyelvek *overload* metódusaihoz) több ágból állhatnak, ahol az ágakat a formális paraméterlista, vagy az ágak

őrfeltételei különböztetik meg egymástól. A *number_of_funclauses* mérték egy függvényt alkotó ágak számát adja eredményül.

2.46. definíció. *[number_of_funclauses] Jelöljük $Fc(f_i)$-vel az f_i függvény ágainak halmazát. Ekkor a mérték eredménye az $Fc(f_i)$ halmaz számossága, vagyis az $NFCL(f_i) = |Fc(f_i)|$.*

A mérték alkalmazható egy $NFCL(f)$, vagy több függvényre ágainak a megszámlálására

$$NFCL(f_1, ..., f_k) = \sum_{j=1}^{k} NFCL(f_i)$$

egyaránt, de mérhetjük vele a modulokban definiált összes függvény ágainak a számát is, ami megegyezik a modul függvényeiben található összes ágak számával, vagyis az $NFCL(m_i) = NFCL(F(m_i))$, a hogy az összes modulra mért eredmény megegyezik az egyes modulokban mért eredmények összegével:

$$NFCL(M) = \sum_{m_i \in M} NFCL(F(m_i)).$$

A függvény ágak megszámlálásához elsőként meg kell keresnünk a függvények definícióját, majd az ágaikat az alábbi útvonal kifejezés bejárásával:

$$n^{(root)} \xrightarrow{\mathbf{seq(module,func,fundef,back,funcl)[1..*]}} [n_1^{clause}, ..., n_m^{clause}]$$

Az eredmény ekkor a gráf bejárása során kapott lista hossza, ami megegyezik a függvény ágak számával. Egy függvényre az útvonalat specializálni kell a mérni kívánt függvény nevével.

2.47. állítás. *Jelöljük a gráf bejárása során kapott listát L-el, amelynek számossága megegyezik a az $\mathcal{SG}$ gráfban található összes függvény függvényágainak számával. Mivel adott esetben egy függvényre mérünk, az útvonalat specializálni kell a függvény nevével és paraméterszámából alkotott feltétellel. Az így kapott listát jelöljük A'-vel. Ebben az esetben az A' lista hossza adja a number_of_funclauses mérték eredményét. (Az SG szemantikus gráfban a függvény első (egy ág esetén az egyetlen ágát) a függvény definíciójának tekintjük, a mérték kiszámítása során azonban minden ágat klóznak számolunk.)*

McCabe féle ciklomatikus szám. A *mc_cabe* bonyolultság mérték értéke a *Thomas McCabe* által konstruált vezérlési gráfban [48] definiált alapvető útvonalak számával azonos, vagyis azzal, hogy hányféle kimenete lehet egy függvénynek nem számítva a benne alkalmazott további függvények bejárási útvonalainak a számát. A *McCabe* ciklomatikus számot eredetileg a procedurális nyelvek alprogramjainak a mérésére fejlesztette ki *Thomas J. McCabe* [48]. Ez a mérőszám alkalmas a funkcionális nyelvek, így az Erlang modulokban implementált függvények mérésére is. *McCabe* a programok ciklomatikus számát a következőképpen definiálja:

2.48. definíció. *[McCabe-féle ciklomatikus szám] A $G = (v, e)$ vezérlési gráf $V(G)$ ciklomatikus száma $V(G) = e - v + 2p$, ahol p a gráf komponenseinek a számát jelöli, ami megegyezik az erősen összefüggő gráfban található lineárisan összefüggő körök számával [40]. A mértéket a 4 fejezetben kiegészítjük néhány új tulajdonsággal.*

2.49. definíció. *[McCabe] Az f_i függvény ágait (overload változatait) jelölje $fc(f_i)$, és az ágakban található if, valamint case kifejezések ágait jelölje $if\_cl(f_i)$, és $case\_cl(f_i)$. Ekkor a McCabe ciklomatikus szám Erlang függvényekre mért eredménye $MCB(f_i) = |fc(f_i)| + |case\_cl(f_i)| + |if\_cl(f_i)|$.*

A mértéket alkalmazhatjuk függvények egy csoportjára

$$MCB(f_1, ..., f_k) = \sum_{j=1}^{k} MCB(f_j).$$

Az $m_i \in M$ modul függvényein mért eredmény megegyezik a modul összes függvényén mért értékek összegével:

$$MCB(m_i) = MCB(F(m_i))$$

Az összes modulra mért érték pedig a modulokra mért értékek összege:

$$MCB(M) = \sum_{m \in M} MCB(F(m))$$

Az eredmény kiszámításához a szemantikus gráfban meg kell keresnünk az adott függvény ágait (*clauses*),

$$n^{(root)} \xrightarrow{\text{seq(module,func,(fundef,back),funcl)[1..*]}} [n_1^{clause}, ..., n_m^{clause}]$$

majd az ágakban található if, valamint *case* kifejezéseket. A kifejezések gyűjtéséhez elsőként a felső szintű kifejezéseket kell megtalálnunk a (*visib*) éleken. (Ezek a kifejezések a függvény legkülső blokkjában találhatóak.) Ezekből elindulva össze kell gyűjtenünk a beágyazott if és *case* típusú kifejezéseket is (beleértve a felső szintű $if, case$ kifejezéseket).

$$n^{(clause)} \xrightarrow{seq(visib,sub\_expr \wedge collect(if\_expr,case\_expr))[1..*]} [n_1^{expr}, \ldots n_l^{expr}]$$

A függvény ágakra azért van szükség, mert ezek is elágazásnak tekinthetőek, így az ágak a függvény lehetséges kimeneteinek számítanak. Ezt bizonyítja az a tény is, hogy a függvény ágakat kiválthatjuk az ágak egyikében elhelyezett *case* kifejezéssel (lásd: 4. fejezetben). Az *if* és a *case* kifejezések ágaihoz szintén el kell jutnunk a $top, \{visib, back\}$ éleken és a darabszámukat hozzáadni az eredményhez.

$$n^{(expr)} \xrightarrow{top,\{visib,back\})[1..*]} [n_1^{expr}, \ldots n_l^{expr}]$$

2.50. megjegyzés. *A mért modulban meghívott függvények akárhány kimenettel (visszatérési pont) rendelkeznek is, egy útvonal végének számítanak, ezért ezeket nem vesszük figyelembe.*

2.51. megjegyzés. *A mérték definícióját és a kiszámításának módszerét a 4. fejezetben kiegészítjük.*

Függvényre irányuló hívások száma. A *calls_for_function* mérték az adott függvényre történő hívások számát adja vissza. Ez a forrásszöveg jellemzése szempontjából nagyon fontos érték, mivel ez alapján lehet eldönteni, hogy egy függvény egy *inteface* modul valamely számításokat végző függvénye, vagy egy *library* modul olyan segéd függvénye, amelyet más modulok használnak egy adott részfeladat elvégzésére. A kiszámítás módja hasonlít a modulra irányuló függvény hívások számának kiszámítására, de itt a modul egyetlen függvényét mérjük és a modulon belülről induló hívásokat is figyelembe vesszük.

2.52. definíció. *[calls_for_function] Valamely f_i függvényre irányuló függvényhívások száma* $CFF(f_i) = |C(F(M), f_i)|$.

A mérték alkalmazható függvények egy csoportjára

$$CFF(f_1, ..., f_k) = \sum_{j=1}^{k} CFF(f_j).$$

Az adott függvényre irányuló hívások kiszámításához elsőként meg kell találnunk a modulokat a *module* éleken keresztül, majd a mérés tárgyát képező függvényt a hozzá vezető *func* élen át. A kereséshez ezen a ponton a gráf útvonal modulból induló részét specializálni kell a függvényre.

Végül a csomópontjából induló $\{funcall, back\}$ éleken át megkaphatjuk a függvényre vonatkozó hívásokat tartalmazó további n^{func} csomópontokat.

$$n^{(func)} \xrightarrow{\{\mathbf{funcall,back}\}[1..*]} [n_1^{(func)}, \ldots, n_t^{(func)}]$$

2.53. állítás. *Az fenti útvonal bejárásának eredménye az összes, az f_i függvényt hívó függvények csomópontjainak listája. Ezt jelöljük L-el. Ekkor az L lista számossága megegyezik a a calls_for_function mérték eredményével.*

Függvényből kiinduló hívások száma. A *calls_from_function* mérték az adott függvényből induló függvény hívások számát méri, vagyis azt, hogy a függvény hány másik függvényt hív meg. Amennyiben egy függvényt kétszer hív a mért függvény, a kapcsolatot egynek számoljuk, és a rekurzív hívásokat sem vesszük figyelembe.

2.54. definíció. *[calls_from_function] Valamely f_i függvényből kiinduló függvényhívások száma $CFMF(f_i) = |C(f_i, F(M))|$.*

A mérték alkalmazható függvények egy csoportjára

$$CFMF(f_1, ..., f_k) = \sum_{j=1}^{k} CFMF(f_j)$$

.

A mérték kiszámításához meg kell keresnünk az $\mathcal{SG}$ gráfban a modulokat a *module* éleken keresztül, majd a mérés tárgyát képező függvényt a hozzá vezető *func* élen át. A kereséshez a gráf útvonal modulból induló részét specializálni kell az adott függvényre. Végül a $\{funcall, back\}$ éleken át megkaphatjuk a modul függvényeire vonatkozó hívásokat tartalmazó függvény csomópontokat.

$$n^{(func)} \xrightarrow{\mathbf{funcall}[1..*]} [n_1^{(func)}, \ldots, n_t^{(func)}]$$

Függvény kifejezések száma. A *number_of_funexpr* mérték a modulban szereplő (*function expression*) függvény kifejezések számát adja vissza. A függvény kifejezés hívását nem, de a bevezetését (definíció) méri. A 2.8 forrásszövegben szereplő függvény kifejezések száma a látszat ellenére egy. Az Erlang programok bonyolultsága mellett azok olvashatóságát, valamint továbbfejleszthetőségét is nagyban befolyásolja az alkalmazott függvény kifejezések és λ függvények száma mind a forráskódot, mind a szintaxisfát tekintve.

Függvény kifejezések

```
...
1. F = fun(A) -> A + 1 end,
2. F(1),
3. F2 = fun a/1,
...
```

2.8. ábra. Függvény kifejezések száma

2.55. megjegyzés. *A függvény kifejezések következő két formáját tudjuk detektálni a forráskódot reprezentáló szemantikus gráfban (az első forma a 2.8 forrásszöveg első sorában elhelyezett, F változóba kötött kifejezésnek, a második az F2 változóba kötöttnek felel meg):*

$fexp_1$:
fun$(p_1^1, \ldots, p_n^1)$ *when* g_1 $\rightarrow$
$e_1^1, \ldots, e_{l_1}^1$;
$\vdots$
$(p_1^m, \ldots, p_n^m)$ *when* g_m $\rightarrow$
$e_1^m, \ldots, e_{l_m}^m$

$fexp_2$:
fun $m : g/n$ *or* fun g/n,

ahol a függvény kifejezéseket a fun nyelvi kulcsszó be, a $p_j^i \in P$ *paraméterek,* $g_i \in G$ *őrfeltételek és* $e_i \in E$ *kifejezések.*

2.56. definíció. [*number_of_funexpr*] *Jelölje az* f_i *függvény által tartalmazott függvény kifejezéseket* $F^{exp}(f_i) = \{f_1^{exp}, ..., f_n^{exp}\}$, *mely halmaz nem tartalmazza az* f_i *függvény által hívott más függvényekben található függvény kifejezéseket. Ez esetben a number_of_funexpr mérték* $NFE(f_i) = |F^{exp}(f_i)|$.

A mértéket alkalmazhatjuk több függvényre: $NFE(f_1, ..., f_k) = \sum_{i=1}^{k} NFE(f_i)$ és modulokra egyaránt. A modulokra mért eredmény megegyezik a modulokban definiált függvényekre mért eredmények összegével: $NFE(m_i) = NFE(F(m_i))$. Az összes modulra mért érték azonos a modulokra mért értékek összegével:

$$NFE(M) = \sum_{m \in M} NFE(m).$$

A függvény kifejezések megszámlálásához a modul típusú csúcsokból kiindulva el kell jutnunk a függvényekig, majd a függvények ágaihoz a $\{fundef, back\}$, $funcl$ éleken keresztül, végül a függvény ágakban meg kell keresni az fun_expr típusú kifejezéseket (a más típusú kifejezésekbe ágyazottakat is).

$$n^{(funcl)} \xrightarrow{par(seq(visib,sub\_expr \wedge collect(fun\_expr)))[1..*]} [[n_1^{expr}, \ldots n_l^{expr}]_1, ...]$$

2.57. állítás. *A fenti gráfútvonal bejárásának eredménye a függvény kifejezések listája, ezt jelöljük L-el. Ekkor az L számossága megegyezik a number_of_funexpr mérték eredményével.*

Üzenetküldések száma. A *number_of_mess_pass* mérték egy függvény esetén az abban található üzenetküldéseket megvalósító kódrészletek, modul esetén a modulban szereplő összes függvényben előforduló üzenetküldések számát méri. Az Erlang támogatja az elosztottságot és az elosztott programokban az adatcserét közös memória használata nélkül valósítja meg. A függvényekből induló, vagy másképpen a modulok közti szinkron, vagy asszinkron üzenetküldések száma ezért nagyban befolyásolhatja a mért program bonyolultságát.

2.58. megjegyzés. *Az Erlang nyelvben az üzenet küldéshez a ! operátort, a fogadáshoz a receive kulcsszót használhatjuk.*

r_0:

e_1 ! e_2

r':

receive

p_1 when $g_1 \rightarrow e_1^1, \ldots, e_{l_1}^1$;

$\vdots$

p_n when $g_n \rightarrow e_1^n, \ldots, e_{l_n}^n$

after

$MS \rightarrow e_1^{n+1}, \ldots, e_{l_{n+1}}^{n+1}$

end

ahol az e_1 ! e_2 az e_1 kifejezés küldése az e_2 kifejezésben adott cél számára. Az szinkron üzenetküldéseknél a fogadó félnél az r' az üzenet fogadását teszi lehetővé. A receive ágaiban a $p_i \in P$ minták, amelyekre az üzenetben kapott kifejezés illeszkedhet, a $g_i \in G$ hasonlóan a case ágainál látottakhoz őr feltételek. $e_j^i \in E$ kifejezések az ágakban. Az üzenet fogadásához tartozik egy olyan ág, amelyet az after kulcsszó vezet be és ez alapértelmezett ágként akkor következik, ha az üzenet az előző ágakra nem illeszkedett, esetleg az MS paraméterben megadott időkorlát letelt. A mérték az üzenetküldések számát méri, vagyis az e_1 ! e_2 alakú kifejezéseket a függvényben.

2.59. definíció. *[number_of_mess_pass] Az f_i függvény tartalmazhat üzenetek küldését megvalósító, e_1 ! e_2 alakú kifejezéseket, ezeket jelölje $S(f_i)) = \{s_1, ..., s_n\}$.*

Ekkor a number_of_mess_pass mérték eredménye a

$$NOMP(f_i) = |S(f_i)|$$

A mértéket alkalmazhatjuk függvények halmazára

$$NOMP(f_1, ..., f_k) = \sum_{i=1}^{k} NOMP(f_i),$$

és modulokra egyaránt. A modulokra mért eredmény megegyezik a bennük definiált függvényekre mért eredmények összegével

$$NOMP(m_i) = NOMP(F(m_i)).$$

Az összes modulra mért érték azonos a modulokra mért értékek összegével

$$NOMP(M) = \sum_{m \in M} NOMP(m).$$

Az üzenetküldések megszámlálásához az $\mathcal{SG}$ gráfban a modulokból el kell jutnunk a függvényekig, majd a függvények ágaihoz a $\{fundef, back\}, funcl$ éleken keresztül, végül a függvény ágakban meg kell keresni az $send_expr$ típusú kifejezéseket (a beágyazottakat is).

$$n^{(funcl)} \xrightarrow{par(seq(visib,sub\_expr \wedge collect(send\_expr)))[1..*]} [[n_1^{expr}, \ldots n_t^{expr}]_1, \ldots]$$

2.60. megjegyzés. *Egy függvényre mérve gráfútvonalat specializálni kell a függvény nevéből és paraméterszámából álló párral.*

OTP modulok használata. Az *otp_used* mértéke a modulban használt *erlang OTP callback*, vagyis az erlang nyelv *Open Telecom Platform* [32] kiterjesztésében szereplő modulok számát adja. Ezt a mértéket nem definiáljuk külön, mivel a mérése teljesen megegyezik az üzenetküldő kifejezések mérését végző mértékével annyi különbséggel, hogy itt a mért modulba behozott OTP modulokat reprezentáló nyelvi elemek számát vizsgáljuk.

Külső hívások száma. Az *external_calls* a mérésben szereplő függvényre alkalmazott, de más modulokból érkező függvényhívások számát adja vissza úgy, hogy egy függvény felől érkező több hívás is csak egynek számít.

2.61. definíció. [*external_calls*] *Az f_i függvényre irányuló, más modulokból érkező függvényhívások száma* $ECL(f_i) = |C(F(M), f_i)|$.

A mérték kiszámításához meg kell találni a modulon keresztül a mérni kívánt függvényt, majd a függvényből induló $\{funcall, back\}$ éleken át megkapjuk a függvényre vonatkozó hívásokat tartalmazó további n^{func} csomópontokat.

$$n^{(func)} \xrightarrow{\{funcall,back\}[1..*]} [n_1^{(func)}, \ldots, n_t^{(func)}]$$

A korábban a *chesion* mértéknél ismertetett útvonal kifejezés mintájára itt is meg kell keresni a függvény definíciókat tartalmazó modulokat, és csak azokat a $(n_i^{(func)}, n_j^{(mod)})$ párokat megtartani, amelyek második eleme, vagyis a modul nem azonos a hívott függvény moduljával.

Belső hívások száma. A *internal_calls* a mérésben szereplő függvényekre alkalmazott, de a függvényeket tartalmazó modulból érkező függvényhívások számát adja vissza úgy, hogy egy függvény felől érkező több hívás is csak egynek számít. A mérték kiterjesztése a *calls_for_function* mértéknek úgy, hogy kizárólag a modulon belülről érkező hívásokat veszi figyelembe.

A definíció és a kiszámítás módja szinte teljesen megegyezik az *internal_calls* mértéknél tárgyaltakéval annyi különbséggel, hogy itt a hívó és a hívott függvény modulján ak meg kell egyeznie, vagyis mindkét függvény definícióját ugyanaz a modul tartalmazza. A mérték eredménye: $ICL(f_i) = |C(F(m_i), f_i)|$, ahol $m_i \in M$ az f_i függvényt tartalmazó modul.

Modulon kívülre irányuló hívások. Az *external_calls_from* a mérésben szereplő függvényből jövő de más modulokba irányuló függvényhívások számát adja vissza.

A mérték a *function_calls_out* (lásd: a 2.37. definícióban) mértéket specializálja ki úgy, hogy csak egy adott függvényből kiinduló, de annak moduljától különböző modulokba tartó hívások számát veszi figyelembe, vagyis nem a modulra, hanem annak egyetlen függvényére számolja ki az eredményt.

2.62. definíció. *[external_calls_from] Jelölje az $M' = M \setminus \{m_i\}$ az összes, kivéve az f_i függvényt tartalmazó modult. Ekkor az $ECF(f_i) = |C(f_i, F(M'))|$.*

Az eredmény kiszámításához a modul, majd a függvény csomópontokból indulva a *funcall* éleken át megkapjuk az összes hívást végző függvény csomópontjainak listáját.

Ennél a mértéknél azonban az útvonalat egy függvényre kell specializálni a függvény nevéből, valamint a paraméterszámából alkotott párral

$$func, \{name = name(f_i), arity = arity(f_i)\},$$

ahol az f_i a mért függvény. A kapott listában meg kell keresnünk azokat az elemeket, amelyek az m_i modulban vannak definiálva.

A kereséshez a $\{func, back\}$ éleken haladva minden hívást végző függvényhez meg kell annak modulját, majd (f_c, m_j) alakú párokat alkotni belőlük, ahol a párok első eleme a hívó függvény, a második annak modulja. A kapott párok listájából ki kell válogatni azokat, ahol a mért függvény modulja nem azonos a párban szereplővel.

Modulon belülre irányuló hívások. A *internal_calls_from* a mérésben szereplő függvényből kifelé tartó, a függvényt tartalmazó modulba irányuló függvényhívások számát adja vissza úgy, hogy egy függvény felől érkező több hívás is csak egynek számít.

A mérték a *function_calls_in* (lásd: a 2.36. definícióban) mértéket terjeszti ki úgy, hogy csak egy adott függvény felé tartó, de annak moduljától különböző modulokból jövő hívások számát veszi figyelembe, vagyis az útvonal kifejezést specializálja egy adott függvényre (nem a modulra, hanem annak egyetlen függvényére számolja ki az eredményt).

2.63. definíció. *[internal_calls_from] Jelölje az $M' = M \setminus \{m_i\}$ az összes, kivéve az f_i függvényt tartalmazó modult. Ekkor az $ICF(f_i) = |C(F(M'), f_i)|$.s*

Az $n^{(root)}$ elemből indulva eljuthatunk a mért függvény csomópontjához, ha az élhez kapcsolt szűrőben megadjuk a keresett függvény nevét

$$func, \{name = name(f_i), arity = arity(f_i)\}.$$

Végül a $\{funcall, back\}$ éleken át megkapjuk a függvényre vonatkozó hívásokat tartalmazó n^{func} csomópontokat:

$$n^{(func)} \xrightarrow{\{\text{funcall,back}\}[1..*]} [n_1^{(func)}, \ldots, n_t^{(func)}] \ .$$

Az eredmény az összes, az f_i-t hívó függvények csomópontjainak listája. A listából ki kell szűrnünk az m_i modulból jövő hívásokat tartalmazó függvényeket.

A szűréshez a $\{func, back\}$ éleken haladva minden hívást végző függvényhez meg kell keresni annak modulját, ,ajd (f_k, m_j) alakú párokat alkotni belőlük, ahol a párok egyik eleme a függvény, a másik annak modulja.

2.64. állítás. *A párok listájából ki kell válogatni azokat a függvényeket, ahol a párok második eleme nem egyezik meg a mért függvény moduljával.*

$$\{(n_i^{(func)}, n_i^{(func)} \xrightarrow{\{\text{func,back}\}} n_j^{(mod)}) | n_i^{func} \in F(M), n_j^{(mod)} \in M\}$$

A fenti útvonal kifejezés és az annak szűrésével kapott listát jelöljük L-el. Ekkor az L számossága megegyezik az internal_calls_from mérték eredményével.

Azonosító hossza. A *length_of_name* mérésben szereplő azonosító (függvénynév) karakterekben mért hosszát adja, ami akkor lehet érdekes, ha a kódsorok és ezáltal a programszöveg olvashatóságát próbáljuk mérni és javítani.

2.65. megjegyzés. *Az id_i az $f_i \in F(M)$ függvényt azonosítja. A hossza az id_i-t alkotó jelek száma, vagyis $LON(f_i) = |id_i|$.*

A függvények nevéhez a szemantikus gráfban a modulokon keresztül juthatunk el. A függvényekről, ahogy minden $n \in N$ gráf csomópontról le tudjuk kérdezni az adott típust jellemző attribútumok értékét. A függvényeket a névvel és a paraméterszámmal azonosíthatjuk, ezután meg tudjuk állapítani a név, vagyis az azonosító hosszát.

2.66. megjegyzés. *Ennek a mértéknek az egyszerű lekérdezésekben nincs jelentősége, mivel ezekben a névvel azonosítjuk a függvényt, így a név hossza adott, viszont a bonyolultság alapú szkriptek futtatása során feltételként szerepelhet (lásd: 4. fejezetben).*

Paraméterek száma. A *number_of_funpars* a mérésben szereplő függvény formális paraméterlistájának az elemszámát (aritás) adja eredményül.

A mérték kiszámításához az adott függvény paramétereinek a számát nem kell direkt módon megszámolni, elég csak megkeresni a függvény definícióját, amely tartalmazza az őt jellemző attribútumokat.

A kapott listából ki kell választani a mérésre szánt függvényt, és megnézni az aritását (a csomópontok attribútumainak a kiolvasására használható függvény az elemző és gráfbejáró algoritmus része).

Ezt az információt a szintaxisfát *(AST=Abstract Syntax Tree)* és abból a szemantikus gráfot (lásd: 2. fejezetben) felépítő elemző algoritmus adja hozzá a függvény csomópontjához.

2.67. megjegyzés. *Az Erlang függvények definíciójában a $p_i \in P$ minták alkotják az adott függvény paraméterlistáját, a nyelv szintaxisa alapján az f_j függvény minden f_i^c függvény ága megegyező számú paraméterrel rendelkezik, valamint p_i az f_j függvény bármely ágának paraméterlistáját reprezentáló mintában szereplő elemeket tartalmazó lista, akkor a mérték eredménye, vagyis $NOFP(f_j) = |p_i|$.*

Függvény összesített értéke. A *function_sum* mérték a függvényre, vagy függvényekre jellemző komplexitási mértékekből számított érték. Az eredmény kiszámításához szükséges mértékek felsorolással megadhatóak.

2.68. definíció. *[functions_sum] Jelölje a RefactorErl bonyolultságot elemző algoritmusával mérhető és ezek közül a a függvényekre alkalmazható bonyolultsági mértékek véges halmazát $Me^{func} = \{me_1^{func}, me_2^{func}, ..., me_n^{func}\}$, valamint legyen $S \subset Me^{func}$.*

Ekkor az

$$FS(S, f_i) := \sum_{me^{func} \in S} me^{func}(f_i)$$

az f_i függvényt jellemző és az S-ben megadott mértékek összesített értéke.

Modult jellemző összesített érték. A *module_sum* a modul függvényein mért (kiválasztott) bonyolultsági mértékek összege. A használni kívánt mértékek listába rendezve alkalmazhatóak a kívánt számban és sorrendben.

2.69. definíció. *[module_sum] Jelölje a RefactorErl bonyolultságot elemző algoritmusával mérhető bonyolultsági mértékek véges halmazát és ezek közül a modulokra alkalmazható mértéket $Me = \{me_1, me_2, ..., me_n\}$, és legyen $S \subset Me$. Ekkor az*

$$MS(S, m_i) := \sum_{me \in S} me(m_i)$$

az m_i modult jellemző, az S-ben megadott mértékek összesített értéke és

$$MS(S, M) := \sum_{m \in M} MS(S, m)$$

az összes modulon mért S-ben adott bonyolultsági mértékek összesített értéke.

2.70. megjegyzés. *A modult jellemző összesített érték nem csupán összegzések eredményeként állhat elő. A kiszámítást végző függvény paraméterezhető számokon értelmezett operátorral $MS(S, m_i, op)$, ahol az operátor lehet sum, amely az alapértelmezett összegzést végzi, vagy lehet avg, az átlagoláshoz és egyéb a méréshez kidolgozott függvény. Az operátor lehet egy olyan Erlang függvény kifejezés (λ függvény) is, amelynek az első paramétere a kiszámított mértékek eredményét tartalmazó lista kell, hogy legyen és ez a függvény a benne megfogalmazottak alapján kiszámolja a végeredményt:*

$$MS(S, m_i, (fun([H|T], ...) \rightarrow ... ; fun([], ...) \rightarrow ... end.))_{\square}$$

Természetesen a mérték ilyen formában történő alkalmazása kizárólag a bonyolultságot elemző algoritmusban érhető el az erre a célra kifejlesztett Erlang nyelvű interfészen keresztül. Ez a lehetőség adott a function_sum mérték esetében is.

2.2.5. Összefoglalás

A program bonyolultságának a vizsgálatakor a forrásszöveg azon tulajdonságait vettük alapul, amely tulajdonságok segítségével képet kaphatunk az Erlang forrásszöveg felépí-

téséről, karakterisztikájáról és a programelemek egyedi, valamint a program egészének teljes bonyolultságáról.

A fejezetben megvizsgáltuk a funkcionális nyelvek azon jellemzőit, amelyek alapján kialakíthatjuk azt az összetett mértékrendszert, amely a későbbiekben segítségünkre lesz a bonyolultság alapú hibadetektálás (lásd: 3. fejezet), a szintén ugyanezen alapokon nyugvó, automatikus forráskód transzformációk kivitelezésére használható nyelvnek (lásd: 4. fejezet), valamint elemző algoritmusának kifejlesztésében.

Megkerestük azokat a szoftver bonyolultsági mértékeket, amelyek az objektum orientált, valamint a procedurális nyelvek strukturális bonyolultságát jól mérik. Alkalmazhatóak funkcionális programok, így Erlang programok mérésére is. Ismertetésre kerültek az átvett bonyolultsági mértékek erlang nyelvű konstrukciók mérésére alkalmas változatai, valamint az Erlang nyelv speciális elemeinek mérésére kidolgozott új bonyolultsági mértékek.

A mellékletekben a bonyolultsági mértékek és az azokat megvalósító függvények ellenőrzésének módszereit is megtaláljuk.

2.3. Szöveges lekérdező nyelv

2.3.1. Bevezetés

A strukturális bonyolultsági mértékek kiszámítása, valamint az eredmények megjelenítésé a korábban bemutatott útvonal kifejezések használatával körülményes, mivel az adatok tárolására használt adatstruktúra részletes ismerete szükséges hozzá. Másrészt, a lekérdezéseket végző algoritmus is túl bonyolult ahhoz, hogy egy lehetséges felhasználói interfész számára megfelelő szolgáltatásokat legyen képes nyújtani. Mindezért szükségünk lehet arra, hogy a bonyolultsági mértékek mérésére alkalmazható útvonal kifejezéseket egy magas szintű nyelv segítségével fejezhessük ki. A magas szintű nyelvnek a 2. fejezetben ismertetett szemantikus gráfon ($\mathcal{SG}$) értelmezett útvonal kifejezéseknél lényegesen magasabb absztrakciós szinten kell lehetővé tennie a bonyolultsági mértékek lekérdezését, vagyis el kell takarja az alacsony szintű gráf műveleteket, és azokhoz szorosan kapcsolódó, útvonalakat leíró függvénykifejezéseket.

2.3.2. A fejezetben ismertetésre kerülő eredmények

A fejezetben bemutatjuk azt az általunk kidolgozott strukturális lekérdező nyelvet, amely használatával a $\mathcal{SG}$ gráfban tárolt programszöveg bonyolultságának kiszámítását végző lekérdezéseket az útvonal kifejezéseknél magasabb absztrakciós szinten el lehet végezni. Az elemző algoritmusban lehetséges összes lekérdezés eredményét megkaphatjuk a használatával.

2.3.3. A strukturált lekérdező nyelv használata

A következő példa bemutatja, hogyan támogatja a lekérdező nyelv a bonyolultság méréséhez szükséges lépések megírását úgy, hogy a forrásszöveget tartalmazó adatbázisban történő navigálással, és a mérések elvégzésévének részleteivel nem kell törődnünk. (az adatbázis alatt a forrást tároló szemantikus gráfot értjük, lásd: 2. fejezetben).

Vizsgáljuk meg a 2.9 forrásszöveget, amely az egyszerűség kedvéért csak néhány függvényt tartalmaz, amelyek a gyorsrendezés algoritmusát valósítják meg.

Gyorsendezés

```
-module(a).

quicksort([H|T]) ->
   {Smaller_Ones,Larger_Ones} = split(H,T,{[],[]}),
   lists:append(quicksort(Smaller_Ones),
                      [H|quicksort(Larger_Ones)]);
quicksort([]) -> [].

split(Pivot, [H|T], {Acc_S, Acc_L}) ->
   if Pivot > H -> New_Acc = {[H|Acc_S], Acc_L};
      true      -> New_Acc = {Acc_S, [H|Acc_L]}
   end,
   split(Pivot,T,New_Acc);
split(_,[],Acc) -> Acc.
```

2.9. ábra. Erlang példaprogram a lekérdező nyelv használatához

A 2.10 szövegben találhatunk egy példát, amely a forrásszövegben látható, *a* nevű modulban szereplő függvények visszatérési pontjait adja eredményül a szöveges *interface* számára (az *interface* jeleníti meg a mérés eredményét).

Láthatjuk, hogy a nyelv struktúráját tekintve hasonlóságot mutat az *SQL* alapú lekérdező nyelvekhez, de a funkcióját tekintve teljesen eltér azoktól. Míg az *SQL* alapú nyelvek relációs adatbázisok tartalmának a lekérdezését, valamint az adatok manipulálását teszik lehetővé, az (*Erlang Metric Query Language*) a strukturális bonyolultsági mértékek lekérdezését segíti egy a forrásszövegből felépített speciális gráfon *SG* (lásd: 2 fejezetben). Ez a gráf szintén relációs adatbázisban tárolódik, de a gráf lekérdezése és manipulációja a relációs adatbázis kezelő felett helyezkedik el és a lekérdezések futtatásához annak szolgáltatásait használja csak fel. Mindezek alapján a mértékek lekérdezése egy három rétegű modellben valósul meg:

1. Legfelső réteg a lekérdező nyelv futtatására alkalmas, amely használja az alacsony szintű útvonal kifejezéseket.

2. Útvonal kifejezések rétege, amely szolgáltatásokat biztosít a lekérdező nyelv számára. A lekérdezések kivitelezéséhez a relációs adatbázis kezelő szolgáltatásait használja fel.

3. Relációs adatbázis kezelő (DBMS), amely szolgáltatásokat nyújt az útvonal kifejezések szintjének, valamint tárolja a programelemeket leíró rekordokat.

A magas szintű lekérdező nyelv alkalmas a bonyolultsági mértékek lekérdezésére a megadott szemantikus gráf csomópontokról (itt a forrásszövegben szereplő programkonstrukciók gráfbeli reprezentációjáról van szó), valamint rendelkezik beépített szűrő feltételekkel.

Létezik a nyelvnek egy olyan kiterjesztése, amely a szemantikus gráfban tárolt programkódok strukturális bonyolultsági mértékeinek a manipulálására szolgál. Ez a kiterjesztett változat a forráskód struktúráját automatikusan transzformáló metaprogramok írását teszi lehetővé. A nyelv használatával konstruált programok alkalmasak arra, hogy a bonyolultság mérése mellett az *SG* szemantikus gráf transzformációjával, előre definiált szabályok betartása mellett átalakítsák a forrásprogram szerkezetét. (Ezt a módszert 4. fejezetben részletesen ismertetjük.)

Gyorsrendezés

```
show fun_return_points for function
         ({'a','quicksort',1}, {'a','split',3}) sum
```

2.10. ábra. Függvény visszatérési pontjainak lekérdezése

Térjünk vissza a 2.10 lekérdezéshez. A *query* elején a *show* szócska arra utasítja az algoritmust, hogy a lekérdezés eredményeket jelenítse meg.

A *fun_return_points* az a bonyolultsági mérték, amelyet le kell kérdezni a forráskód, vagyis a szemantikus gráf azon részén, amelyet a lekérdezés második felében definiálunk.

Az elemző algoritmusban minden bonyolultsági mérték függvényként van megkonstruálva, amely függvények az adott mérést megvalósító, előre definiált útvonal kifejezéseket tartalmazzák (lásd: a 2. fejezetben).

A lekérdezés következő szakaszában található az annak tárgyát képező csomópont típusa, amely jelen esetben *function*, vagyis függvény. Az itt szereplő elemek egy (vagy több) függvény, vagy modul szemantikus csomópontjai lehetnek a forrásszöveg tárolására használt szemantikus gráfban és a csomópontok típusa meg kell, hogy egyezzen az előbbiekben definiált típussal (*function*).

A példában azokat a függvényeket soroljuk fel, amelyekre a lekérdezést futtatjuk. A függvényeket az őket tartalmazó modul nevéből, az adott függvény nevéből és az

aritásából (paraméterek száma) összeállított hármassal adjuk meg (pl.: *{a,quicksort,1}*, *{a,split,3}*, ...).

A lekérdezés végére feltételeket helyezhetünk el, amelyek segítségével a kapott eredmény tovább szűrhető.

Láthatjuk, hogy a nyelv használata nagyon kényelmessé teszi a bonyolultsági mértékek mérését, valamint azok kombinálását és szűrését a különböző csomópont típusokra. Mindezek mellett viszonylag egyszerű és könnyen bővíthető mind a szintaxis, mind az alkalmazható mértékek, valamint a szűrő feltételek listája.

Valójában az új mérési módszerek nem is a nyelv szintaxisának szintjén építhetők az algoritmusba, hanem az alacsony szintű útvonal kifejezéseknél. Ez azért lehetséges, mert a nyelv dinamikája lehetővé teszi az útvonal kifejezések (és a segítségükkel mérhető bonyolultsági mértékek) bővítést úgy, hogy a szintaxist nem kell megváltoztatni.

Ez a gyakorlatban azt jelenti, hogy a konkrét lekérdezéseket végző függvények (bonyolultsági mértékek) nevét nem kell a nyelvtanba beépíteni. Ha a függvény neve, paraméterei és a visszatérési értéke a megfelelő formát követik, az automatikusan használható lesz a lekérdező nyelvben.

2.3.4. A nyelv szintaxisa

MetricQuery → **Show for Type**
Show → **show** *metric*
Type → **module** *module* | **function** *function*

2.11. ábra. A lekérdező nyelv szintaxisa

A lekérdező nyelv szintaxisa a 2.11 ábrán látható. A nyelvtan (2.12. lista) viszonylag egyszerű és megfelel az *LALR(1)* [36] nyelvtanok támasztotta követelményeknek. (Az nyelvi elemző *Yecc* [32] parszer generátorral készült, de a generálás folyamatáról és az algoritmusba való beépítés feladatairól nem ejtünk szót.)

A 2.12 lista utolsó sorában találjuk a nyelvhez tartozó szűrőket (*filter* néven), vagyis azokat az azonosító szimbólumokat, amelyek a szűrő, és az eredményt a megfelelő formára alakító függvényeket aktivizálják. Ezeket a lekérdezések végén helyezhetjük el, amelyek a futtatás során előálló eredményeket specializálják - hasonlóan a *UNIX* alapú operációs rendszerek szűrőihez. A szűrést megvalósító függvények némelyike kiválthatná számos alkalmazott mérték használatát, amelyekre csak azért van szükség, mert az elemző más részei, mint a szemantikus lekérdezéseket végző algoritmus praktikus okokból nem használhatják a szűrőket

Míg az előző példákban a lekérdezések szerkezetét elemeztük, a következőkben azt vizsgáljuk, hogy az adott *query* az elemző algoritmus szintjén milyen mechanizmusokat indít el.

Lekérdező nyelvtan

```
predicates ::= <predicate>
predicates ::= <predicate> <filter>
predicate  ::= show <metric> for function <funlist>
predicate  ::= show <metric> for module <modlist>
predicate  ::= save metric for function <funlist>
predicate  ::= save metric for module <modlist>
nodetype   ::= function|module
metric     ::= <metricfuns>
modlist    ::= '(' ')'
modlist    ::= '(' mods ')'
mods       ::= mod
mods       ::= mod ',' mods
mods       ::= string
funlist    ::= '(' ')'
funlist    ::= '(' funs ')'
funs       ::= fun
funs       ::= fun ',' funs
fun        ::= '{' fune '}'
fune       ::= ftag1 ',' ftag2 ',' ftag3
ftag3      ::= integer
ftag2      ::= string
ftag1      ::= string
filter     ::= sum|max|avg|tolist|totext|fmaxname
```

2.12. ábra. Lekérdezések nyelvi konstrukciói

A komplexitási mértékek lekérdezését végző algoritmus futtatásához kifejlesztett infrastruktúra két szintre osztható fel. Az alsó szint az egyes mértékek kiszámítását lehetővé tevő függvényeket tartalmazza, valamint azokat az *interface* függvényeket, amelyek a szemantikus gráfon ($\mathcal{SG}$ lásd: 2 fejezetben) elvégzik a megfelelő méréseket. A magasabb szint a lekérdező nyelv szintaxisát ellenőrző lexikális és szintaktikai elemző, valamint a lekérdezések Erlang nyelvű felület számára érthető formára való leképezést végző részekből áll.

Lekérdezés modulra

```
show number_of_fun for module ('a','b')
```

2.13. ábra. A modul függvényeinek száma

A két logikai egység külön-külön is használható a mértékek lekérdezésére, vagyis hívhatjuk a függvényeket tartalmazó modult Erlang nyelvű programokból és közvetlenül az *erlang shell*-ből is.

Egy tetszőleges lekérdezést elsőként a lexikális elemző, majd a szintaktikus elemző dolgozza fel, vagyis a lekérdezés szövegéből előállítja azt a formát, amelyet a bonyolultsági mértékeket implementáló függvények fel tudnak dolgozni.

Vizsgáljuk meg a 2.13 programlistában látható lekérdezést, amely megszámolja a megadott modulban található függvények számát.

A lekérdezésben a *show* előtagot követi *number_of_fun* címke, ami a kiszámításra váró mértéket reprezentáló függvény lekérdezésben alkalmazható neve.

A *module* a csomópontok típusa, amelyekre a lekérdezést futtatjuk, és az *{a, b}* rendezett pár a lekérdezésben szereplő, vagyis a mérés tárgyát képező modulok neveit tartalmazza.

Lekérdezés modul függvényeire

```
show branches_of_recursion for
        function ({'a','f',1},{'a','g',0}) sum
```

2.14. ábra. Függvény rekurzív hívásainak a száma

Amennyiben a csomópont típusát *function* típusúra állítjuk, a listának a következő elemeket kell tartalmaznia: a modul nevét, melyben az adott függvényt definiálták, a függvény nevét és a függvény paramétereinek a számát (több függvény esetén többször ugyanezt). Ezek alapján tehát a *({test,f,1})* elemekkel egy függvényt írunk le, amely a *test* modulban van definiálva, *f* a neve és *1* a formális paramétereinek a száma.

A függvényekre való hivatkozást a 2.14 programlistában láthatjuk, ahol az *a* nevű modulban definiált függvényekhez szeretnénk meghatározni a rekurzív függvényágak számát, majd a lekérdezés végén elhelyezett szűrővel (*sum*) a kapott értékeket összegezni (a *sum* nélkül minden függvényhez külön-külön megkapnánk a rekurzív ágak számát).

Amennyiben nem alkalmazunk szűrőt az eredmény (lásd: a 2.15. ábrán) tartalmazza a mérésben szereplő bonyolultsági mérték nevét, a hozzá tartozó (mért) értéket, valamint a csomópontokat, amelyekre mértünk.

Az elemző algoritmus kigyűjti a csomópont által reprezentált modul nevét, vagy függvény esetén annak modulját, nevét és paramétereinek számát, ha erre szükség van (és erre utasítjuk). Ezeket az adatokat már csak formáznia kell. (Jelenleg az implementált, aktuális verzióhoz az *emacs* nevű program nyújtja a felhasználói felületet, és *Elisp* nyelven van programozva a megjelenítést végző felület, tehát a csomópontok alapján szöveges formában, de listákba gyűjtve kell visszaadni a mérési eredményeket.)

A szűrő lehet aggregáló függvény, vagy az eredmény struktúrájának a meghatározására használható függvény lekérdezésben szereplő azonosítója. Az alapértelmezett

Lekérdezés eredménye

```
(
  ((a, f, 1),
     (branches_of_recursion, 2)),
  ((a, g, 0),
     (branches_of_recursion, 3)),
  ...
)
```

2.15. ábra. Az elemző számára átalakított lekérdezés

szűrő a *tolist*, ami az eredményt egyszerű listaként adja vissza. Az így kapott listát alakítjuk át a 2.15 példában látható és a felhasználói felület számára érthető formára.

Az aggregáló függvények az aggregátum eredményét adják vissza úgy, hogy a 2.15 listából kigyűjtik a lekérdezések eredményeit (a neveket és egyéb adatokat kihagyják), majd elvégzik azokon a megfelelő műveleteket (sum, avg, stb).

Amennyiben a filter ténylegesen szűr, kiválogatja az eredmény listából a meghatározott feltételnek megfelelő elemeket. Abban az esetben viszont, ha a formátumra nézve tartalmaz utasításokat (pl.: *tostring*), a kimenetet a kért formátumra hozza egy erre a célra kifejlesztett szövegformázó függvény alkalmazásával.

2.3.5. Összefoglalás

A *Szöveges lekérdező nyelv* című fejezetben bemutattuk azt a strukturális lekérdező nyelvet, és a hozzá tartozó elemző algoritmust, amely segítségével az $\mathcal{SG}$ szemantikus gráfba betöltött Erlang programok strukturális bonyolultsága lekérdezhető (lásd: 2. fejezetben).

A fejezet második felében megismerhettük a nyelv szintaxisát, a nyelvi elemek segítségével definiálható lekérdezéseket és a lekérdezésekhez kapcsolható szűrők működését, valamint az alkalmazásukban rejlő előnyöket.

A lekérdező nyelv szorosan kapcsolódik a 2. fejezetben bemutatott útvonal kifejezésekhez, mivel az elemző algoritmusa közvetett úton útvonal kifejezésekre fordítja a lekérdezések szövegét.

A lekérdezések és a lekérdező nyelv működésének alapja, hogy a forrásszövegből létrehozott szemantikus gráf elemeinek sorozataira (lásd: a 2. fejezetben) a strukturális bonyolultsági mértékeket kiszámító függvények meghívhatóak a lekérdezésekben szereplő nyelvi elemek segítségével. A definiált lekérdező nyelv futtatását végző algoritmust funkcionális nyelven (Erlang) valósítottuk meg és elkészítettük a lekérdezések futtatására alkalmas interface -t.

2.4. A tézis megfogalmazása

Erlang nyelvre alkalmazható metrikák kidolgozása, mérése és lekérdező nyelv megalkotása.

Magyarázat. Az $\mathcal{SG}$ kiterjesztésével bevezettünk egy olyan adatstruktúrát, amely a forráskódot jellemző bonyolultsági mértékek elvárt és mért értékei alapján képes nyilvántartani és megjelölni a nehezen kezelhető programrészeket.

A létező szoftver bonyolultsági mértékeket átalakításával, valamint új mértékek kidolgozásával összetett mértékrendszert hoztunk létre annak érdekében, hogy az Erlang nyelv bonyolultságot tekintve releváns tulajdonságait és összes program konstrukcióját mérni tudjuk.

Az $\mathcal{SG}$ gráfon értelmezett útvonal kifejezések kiváltására definiáltunk egy olyan strukturált lekérdező nyelvet, amely magasabb absztrakciós szinten teszi lehetővé a szoftver bonyolultsági mértékek lekérdezését.

2.5. Releváns publikációk

- Horváth, Z., Lövei, L., Kozsik, T., Kitlei, R., Víg, A., Nagy, T., Tóth, M., and Király, R.: *Modeling semantic knowledge in Erlang for refactoring.* In Knowledge Engineering: Principles and Techniques, Proceedings of the International Conference on Knowledge Engineering, Principles and Techniques, KEPT 2009, volume 54(2009) Sp. Issue, Studia Universitatis Babeş-Bolyai, Series Informatica, pages 7–16, Cluj-Napoca, Romania, Jul 2009

- Tamás Kozsik, Zoltán Csörnyei, Zoltán Horváth, Roland Király, Róbert Kitlei, László Lövei, Tamás Nagy, Melinda Tóth, and Anikó Víg. *Use Cases for Refactoring in Erlang.* In Central European Functional Programming School (The Second Central European Summer School, CEFP 2007, Cluj, Romania, June 23-30, 2007), Revised Selected Lectures, volume 5161 of Lecture Notes in Computer Science, pages 250-285. Springer Berlin/Heidelberg, 2008. rev: Zbl 1170.68414, DBLP

- Lövei, L., Tóth, M., Horváth, Z., Kozsik, T., Király, R., Kitlei, R., Bozó, I., Hoch, C., and Horpácsi, D.: *Reengineering legacy Erlang code by refactoring.* Central Europen Functional Programming Summer School, May 2009.

- R. Kitlei, L. Lövei, M Tóth, Z. Horváth, T. Kozsik, T. Kozsik, R. Király, I. Bozó, Cs. Hoch, D. Horpácsi. *Automated Syntax Manipulation in RefactorErl.* 14th International Erlang/OTP User Conference. Stockholm, November 13, 2008.

- Lövei, L., Hoch, C., Köllö, H., Nagy, T., Nagyné-Víg, A., Kitlei, R., and Király, R.: *Refactoring Module Structure* In 7th ACM SIGPLAN Erlang Workshop, 2008
- Zoltán Horváth, László Lövei, Tamás Kozsik, Róbert Kitlei, Anikó Nagyné Víg, Tamás Nagy, Melinda Tóth, and Roland Király. *Building a Refactoring Tool for Erlang.* In K. Mens, M. van den Brand, A. Kuhn, H.M. Kienle, and R. Wuyts, editors, 1st International Workshop on Academic Software Development Tools and Techniques, 2008. 11 pages.
- Lövei, L., Hoch, C., Köllő, H., Nagy, T., Nagyné-Víg, A., Horpácsi, D., Kitlei, R., and Király, R.: *Refactoring Module Structure* In Proceedings of the 7th ACM SIGPLAN workshop on Erlang, pages 83–89, Victoria, British Columbia, Canada, Sep 2008.
- Tóth, M., Bozó, I., Horváth, Z., Kitlei, R., Király, R., Horpácsi, D., and Kőszegi, J.: *RefactorErl: a source code analyser and transformer tool* Poster at the High Speed Network Workshop 2011, Budapest, Hungary, May 2011
- [15] Zoltán Horváth, László Lövei Tamás Kozsik, Roland Király, Melinda Tóth, Róbert Kitlei, Dániel Horpácsi, István Bozó. *Complexity Metrics and simple semantic queries for Erlang.* Report Ericsson Hungary 2009.
- [8] Zoltán Horváth, László Lövei Tamás Kozsik, Roland Király, Melinda Tóth, Róbert Kitlei, Dániel Horpácsi, István Bozó. *Extended semantic queries on Erlang programs and comprehensive testing of RefactorErl.* Tech. Report 2010. Ericsson Hungary 2010.
- Lövei, L., Horváth, Z., Kozsik, T., Király, R., Víg, A., and Nagy, T.: *Refactoring in Erlang, a Dynamic Functional Language* In Proceedings of the 1st Workshop on Refactoring Tools, pages 4546, Berlin, Germany, July 2007 extended abstract
- Zoltán Hernyák, Roland Király. *Teaching programming language in grammar schools.* Annales Mathematicae et Informaticae 36 (2009) Pages: 163-174
- Király Roland, *Funkcionális programozási nyelvek* EKF TTK TAMOP412 2010 120 oldal.
- Király, R., Kitlei R.: *Complexity measurments for functional code* 8th Joint Conference on Mathematics and Computer Science (MaCS 2010) refereed, and the proceedings will have ISBN classification July 14-17, 2010
- Király, R., Kitlei R.: *Implementing structural complexity metrics in Erlang.* '10 ICAI 2010 – 8th International Conference on Applied Informatics to be held in Eger, Hungary January 27-30, 2010
- Király, R. and Kitlei, R.: *Implementing structural complexity metrics for Erlang* Poster on the 8th International Conference on Applied Informatics, ICAI 2010, 2010

3. fejezet

Bonyolult programrészek lokalizálása

3.1. Bevezetés

Statikus forrásszöveg elemzéssel már a programfejlesztés kezdeti fázisaiban fényt deríthetünk a programban előforduló következetlenségekre, és segítségével megtalálhatjuk és javíthatjuk a kezelhetetlenül bonyolult programrészeket, ezáltal csökkentve a programtesztelés költségeit.

A bonyolultsági mértékeken alapuló elemzés magában foglalja a bonyolult programrészek, a rossz programozási stílusjegyek, valamint a nem hatékony kódrészletek lokalizálását.

Mivel a mérésre használt mértékrendszer, és a mért értékek egymáshoz viszonyított aránya eltérhet a különböző programok fejlesztésénél egy paradigmán belül is, lehetőséget kell biztosítanunk arra, hogy a program tervezői definiálni tudják a számukra megfelelőnek ítélt értékeket és arányokat.

A fejlesztés, és a programok elemzése során meghatározó kérdés, hogy a megfelelő kódminőség eléréséhez, és megtartásához a program mely részeinek a változásait kell figyelemmel kísérni, és milyen tulajdonságokat kell az adott programelem vizsgálata során figyelembe venni.

A fejezetben bemutatásra kerülő modell, és a modell alapján konstruált algoritmus a programkódon mért bonyolultsági mértékek tárolása mellet lehetőséget biztosít arra is, hogy a programok moduljaihoz és függvényeihez definiálhassuk az azokat jellemző bonyolultsági mértékek számunkra megfelelőnek ítélt értékeit.

Az előre definiált, és az aktuálisan mért értékek összehasonlításával (eltérés esetén) automatikusan meg tudja jelölni a rossz minőségű programrészeket - összetett, vagy több mértékből számított bonyolultsági mérőszámok vizsgálata mellett. (Ilyen eltéréseket okozhat, ha a programozó, vagy valamely a forrásszöveget érintő programtranszformációs lépés, megváltoztatja a program szintaxisát, szemantikáját, vagy a lexikális struktúráját.)

A korábban a 2.2.5. fejezetben ismertetett lekérdező nyelv használatával az átalakításokat megelőző, és az aktuális állapotban mért értékek minden változása, eltérése kimutatható, de ez a módszer nagyon körülményes, és nem túl hatékony nagyméretű forrásszövegek elemzésére.

Mindezek mellett a bonyolultsági mértékek aktuális értékeinek ismeretében sem minden esetben dönthető el, hogy a programkód változtatása jó, vagy rossz irányba terelte-e a strukturális bonyolultságot, vagyis, hogy javított-e a mérés tárgyát képező forrásszövegen, vagy éppen rontott a minőségén.

A probléma megértéséhez vizsgáljuk meg a 6. fejezetben bemutatott 6.1 táblázatot, amely a *RefactorErl* forrásszövegének két különböző verzióján mért bonyolultsági mértékek értékeit mutatja. Figyeljük meg a forráskód átalakítása előtti, és az utána mért értékek közötti eltéréseket.

3.1. megjegyzés. *Ahogy korábban említettük, a méréseket akkor végeztük el, mikor a program egy komolyabb átalakításon esett át, amit a forráskód mennyiségének a növekedése indukált. Az újabb és újabb funkciók bevezetése miatt a forrásszöveg kezdett kezelhetetlenné válni. Az átalakítás célja a modularizáció, és a függvényeket tartalmazó modulok átstrukturálása volt.*

A táblázat számait elemezve, és az különböző értékek változásait figyelve az alábbi fontos megállapítást tehetjük: Annak ellenére is, hogy ismertek az átalakítás előtti, és az azt követően mért értékek, nehezen lehet megállapítani, hogy a bonyolultsági mértékeket érintő változások jók, vagy rosszak.

Tekintsük példaként a táblázatban a *coupling* nevű mérték eredményét, amely 932-ről 4968-ra változott a módosítást követően. Ez az adat nevezhető jónak, de könnyen bebizonyíthatjuk ennek az ellenkezőjét is, mivel minden esetben a számunkra, vagy a céljaink szerint legjobbnak vélt értékek figyelembe vételével döntünk. Nyilván ez a legtöbb bonyolultsági mérték esetén ugyanígy van, ami természetesen nem baj, de ilyen körülmények között nem szabad elvenni a döntés lehetőségét a program fejlesztőjétől.

3.2. A fejezetben bemutatásra kerülő eredmények

A fejezetben bemutatjuk azt az elemző algoritmust és a hozzá konstruált szabályrendszert, amely megkeresi, megjelöli és összegyűjti további feldolgozás céljából a programok kezelhetetlenül bonyolult részeit. Ismertetjük az elemző algoritmusnak azt a funkcióját is, amely a bonyolult részek megjelölése mellet a hibásnak vélt gráf csúcsok által reprezentált szakaszokat a forrásszövegben is képes megtalálni, valamint megmutatjuk a különböző program transzformációk forrásszövegre gyakorolt hatásait.

3.3. A forrásszöveg bonyolultságának elemzése

Az probléma ismertetését követően nézzük meg, hogyan kell működnie a bonyolult programrészek lokalizálását végző algoritmusnak.

Első lépésként a forrásszövegből fel kell építenünk a program szintaxis fáját, kiegészíteni azt a statikus analízissel nyerhető összes szemantikus információval, majd meg kell konstruálnunk a program hívási gráfját, és ezt is hozzáadnunk a szintaxisfához (lásd: 2. fejezet). Az így kapott gráfot ki kell egészítenünk a programon mért adatfolyam analízis eredményével, és a forrásszöveget alkotó lexikális elemekkel.

A következő lépés a strukturális bonyolultsági mértékek kiszámítása és tárolása a szemantikus gráfban (pontosabban annak a bonyolultságot tartalmazó részében). A bonyolultsági mértékek tárolására tehát a szemantikus gráfban azzal konzisztensen egy feszítőfát kell nyilván tartanunk, amely nem csak a kiszámolt mértékeket, hanem a fejlesztő által beállított célértékeket is tartalmazza.

Az előre definiált értékek tárolására azért van szükség, hogy legyen mivel összehasonlítani az aktuális transzformációs lépést követően kiszámolt bonyolultsági mértékeket.

Mikor a gráf elkészült, minden csomópontra alkalmaznunk kell egy a mért, valamint a beállított értékek összehasonlítását végző függvényt, majd jelezni kell az eltéréseket.

A bonyolultsági mértékek többszöri mérése során fontos, hogy a programszöveget reprezentáló $\mathcal{SG}$ gráfnak (lásd: a 2. fejezetben) megtaláljuk azt a részét, ahol a bonyolultsági mértékeket mindenképpen mérni kell (nyilván azt is, hogy hol nem kell mérni...) az adott transzformációt követően, valamint azt is, hogy mely mértékeket kell az adott gráf csomópontokhoz újra kiszámítani.

A probléma megoldásához meg kell határoznunk azt, hogy mi a transzformáció *legszűkebb környezete.*

Ez a nem összefüggő részgráfja az $\mathcal{SG}$ gráfnak csak azokat a csúcsokat tartalmazza, amelyekre az adott transzformáció hatást gyakorolt (pontosabban amelyekre hatást gyakorolhat).

A bonyolultsági mértékek változtatásokat követő újraszámolására két lehetőség van. Az első, és kevésbé hatékony változatban *minden, a transzformációban érintett modulra és függvényre újra kiszámítjuk az összes bonyolultsági mértéket.* A modulok érintettsége azt jelenti, hogy az adott szemantikus gráfcsomópont esetén, amelyen a transzformációt elvégeztük, mely más modulok, vagyis más gráf csomópontok vesznek részt az eseményben.

Példaként, ha egy exportált függvényen átnevezést hajtunk végre, az átnevezés érinti a függvényt tartalmazó modult, valamint minden olyan függvényt a modulon belül és kívül, amelyek hívják azt. A teljesség kedvéért nézzünk meg egy másik transzformációt, a változó eliminálását (*eliminate_variable*). A művelet a kijelölt változót az összes

előfordulási helyén lecseréli a változóba eredetileg kötött kifejezésre. Ez a transzformáció a program működését tekintve lokális az őt tartalmazó függvényre, de a függvényt tartalmazó modulra a bonyolultsági mértékeket tekintve hatással van, mivel megnöveli a karakterszámot, a sorok számát, valamint megváltoztatja a szemantikus gráf struktúráját azzal, hogy új kifejezéseket vezet be, és változókat szüntet meg.

Más transzformációk esetén még ennél is bonyolultabb a helyzet, de vannak olyanok is, amelyek csak nagyon kis mértékben, vagy egyáltalán nem hatnak a bonyolultságra. Ilyen transzformáció a paraméterek sorrendjének a megváltoztatása, vagyis a *reorder_function_parameters*. (Igazság szerint ez a transzformáció is változtathat a forrásszövegen, ha nagyon hosszú azonosítókat használunk, és a csere során két hosszú azonosító egymás mellé kerül. Ilyenkor a *Pretty Printer* töri a sort, így megváltozhat a sorok száma, vagyis a *line_of_code* értéke, de ez ritka eset.)

Tehát az érintettség azt jelenti, hogy a bonyolultsági mértékek mely modul és függvény csúcsoknál változnak meg az egyes transzformációkat követően.

A másik lehetőségünk a bonyolultsági értékek változásának a követésére, hogy az első módszer mintájára *megkeressük a transzformált csúcsokhoz kapcsolódó más csúcsokat, vagyis a transzformáció azon környezetét, amely érintett az eseményben*, és elemezzük az elvégzett transzformációs lépés forrásszövegre kifejtett hatásait (ez minden transzformációnál más, de két egymást követő azonos lépés esetén ugyanaz). *Az így kapott információ ismeretében meghatározhatjuk, hogy a transzformáció, valamint a csúcs típusa alapján mely bonyolultsági mértékeket mely csomópontokra kell újra kiszámítani.*

A fentiek alapján tehát szükségünk van egy olyan szabályrendszerre, amely pontosan meghatározza, hogy:

- mely transzformáció alkalmazása esetén,
- a szemantikus gráf mely csomópontjaira,
- melyik bonyolultsági mértékeket kell újra kiszámítani.

A probléma általánosan a következőképp írható le: Adott a csomópontokra alkalmazható transzformációk halmaza $TR = (tr_1, tr_2, ..., tr_n)$, ahol a tr_i a rendszerben elvégezhető i^{th} transzformációs lépés. Adott továbbá a mérhető strukturális bonyolultsági mértékek halmaza $Me = \{me_1, me_2, \ldots, me_l\}$, valamint a szemantikus gráf modul $M = \{m_1, m_2, ..., m_k\}$, és függvény típusú csúcsai $F(M)$, ahol $F(m_i)$ az $m_i \in M$ modulban definiált függvények halmaza. A továbbiakban a modul és függvény csúcsokra n_i formában fogunk hivatkozni, ahol az $n_i \in N$ mind szemantikus gráf csúcsok, a $type(n_i)$ függvény pedig a csúcs típusát adja meg. Kikötés még, hogy a bonyolultsági mértékeket kizárólag függvény, valamint modul típusú gráf csúcsokra értelmezzük. Az adott előfeltételek mellett keressük azt a kétváltozós függvényt amelynek az egyik paramétere

a transzformált csomópont típusa $type(n_i)$, a második paramétere pedig az aktuálisan elvégzett transzformációs lépés tr_i.

$$f(type(n_i), tr_i) \longrightarrow [\{n_1, [me_i, ..., me_j]\}, ..., \{n_l, [me_j, ..., me_k]\}]$$

A függvény visszatérési értéke egy rendezett pár, melynek első eleme az érintett gráf csúcsok halmaza, a másik eleme pedig az újra kiszámításra váró bonyolultsági mértékeké. Ha pontosak akarunk lenni, akkor a visszatérési érték rendezett párok sorozata, ahol a párok első elemei az érintett gráf csúcsok, és a második elem minden esetben az első elemként adott csúcson újra kiszámításra váró bonyolultsági mértékek halmaza. Az f általános értelemben vett függvény, de a paramétereinek értékeiből nem tudjuk közvetlenül kiszámítani az eredményét. Az f egy olyan hozzárendelés, amely a transzformált szemantikus gráf csúcs típusa, és a rajta elvégzett transzformációs lépés alapján egy táblázatból ki tudja választani a bonyolultsági mértékek azon részhalmazát ($Me' \subset Me$), amelyet a transzformált, valamint a transzformációban érintett csúcsokon újra ki kell számítani. Mindegyikhez azokat a mértékeket rendeli, amelyeket azon a csúcson kell kiszámolni ($\{n_i, [me_j..., me_k]\}$).

A hozzárendelések elvégzéséhez szükséges táblázatot a következő alfejezetben ismertetésre kerülő szabályrendszer alapján dolgoztuk ki. A szabályrendszer implementációját nem ismertetjük, mivel annak formája technikai probléma és az implementáció része. A transzformációs lépések hatáselemzését viszont részletesen megvizsgáljuk.

3.4. A transzformációk hatáselemzése

A megfelelő szabályok kidolgozásához sorra kell vennünk a transzformációs lépéseket, és mindkét mért csúcstípushoz (*module*, *function*) meg kell határoznunk, hogy mit tekintünk a csúcs környezetének, és mely mértékeket kell arra kiszámolni. Miután megismertük az adott transzformáció működését, meghatározhatjuk, hogy az hol és mire hat a programot reprezentáló gráfban.

3.4.1. Függvény átnevezése

A *rename_function*, vagyis a függvények átnevezése működését, valamint a strukturális bonyolultságra gyakorolt hatását tekintve az egyik legegyszerűbb transzformáció. A kijelölt függvény nevét változtatja meg úgy, hogy közben az azt hívó függvényekben, valamint a minősítéseknél (*module:function*), és az export listákban kompenzálja a változtatások hatásait. A transzformáció nem változtatja meg az azonos nevű, de más paraméter számú függvények neveit, az azonos nevű rekordok és azok mezőinek a neveit sem. Amennyiben a bonyolultsági mértékek változásait figyelembe véve elemezzük a transzformációt, megállapíthatjuk, hogy az hat a sorok számára

Függvény átnevezése

```
-module(renfun).
-export([osszeg/2]).

osszeg(A, B)
    when is_list(A)->
    A ++ [B].

osszeg(A, B) ->
    A + B.
```

→

Az osszeg/2 átnevezése

```
-module(renfun).
-export([sum/2]).

sum(A, B)
    when is_list(A)->
    A ++ [B].

sum(A, B) ->
    A + B.
```

3.1. ábra. Az `osszeg/2` függvény átnevezése `sum/2`-re.

(*line_of_code*), a karakterek számára (char_of_code), megváltoztatja a maximális sorhosszokat (max_length_of_lines), valamint a *no_space_after_comma* mértéket.

Ezek a változások nem változtatják meg komolyabban a forrásszöveget, de mindenképpen mérni kell őket és a programszöveg változásait követően újra kell számolni az értékeiket. A felsorolásban szereplő mértékeket az adott függvényre, annak moduljára, a függvényt hívó többi függvényre és azok moduljaira is újra ki kell számolni, vagyis a bonyolultság újra kalkulálása szempontjából a feladat elég összetett.

Meg kell keresni a függvény aktuális ágának csomópontját a szemantikus gráfban, majd a megfelelő útvonal kifejezés segítségével el kell jutni a függvényt tartalmazó modulhoz. Ezután a függvény hívásokon haladva meg kell keresni a hívó függvényeket és ezek moduljait is a kiindulási függvénnyel végzett kereséshez hasonlóan. Az ellenőrzési és újraszámítási folyamat, vagyis a szabály alapjául szolgáló algoritmus ezek alapján a következő:

1. Megjegyezzük a transzformációt (ha az *refactoring*, egyébként, nem refactoring alapú, ismeretlen változásként azonosítjuk azt).

2. Megállapítjuk a csomópont típusát, amelyen a transzformáció, vagy az egyéb szemantikus gráfot ért változás történt.

3. A transzformáció, és a csomópont típusa alapján a megfelelő útvonal kifejezések előállításával megkeressük a kapcsolódó függvény és modul csomópontokat.

4. A megtalált csomópontokra újra kiszámítjuk azokat a bonyolultsági mértékeket, amelyeket a transzformáció elemzésénél leírtunk.

5. A szemantikus gráfban a bonyolultsági mértékeket az adott csomópontokhoz tároló részeket frissítjük a legutolsó lépésben kiszámolt értékekkel, és eltérés esetén jelezzük a hibákat.

A transzformációban résztvevő programelemek az átnevezés tárgyát képző függvény, az átnevezés tárgyát képző függvény modulja, a függvényt hívó más függvények, és azok moduljai,

Az újraszámításra kerülő strukturális bonyolultsági mértékek listája a transzformációban résztvevő függvényre nézve:

$(line\_of\_code, char\_of\_code, max\_length\_of\_line,$
$average\_length\_of\_line, no\_space\_after\_comma, function\_sum)$

A transzformációhoz kapcsolódó, de közvetlenül nem transzformált függvényeken újraszámolandó mértékeinek listája:

$(line\_of\_code, char\_of\_code, max\_length\_of\_line,$
$average\_length\_of\_line, no\_space\_after\_comma, function\_sum)$

Az transzformációban érintett modul csomópontok listájára számolandó bonyolultsági mértékek:

$(line\_of\_code, char\_of\_code, average\_size, max\_length\_of\_line,$
$average\_length\_of\_line, no\_space\_after\_comma)$

3.4.2. Elemzés összefoglalása

A függvények átnevezése transzformáció hatást gyakorol a résztvevő függvényre, annak moduljára, a függvényt hívó egyéb függvényekre, valamint ezek moduljaira.

A transzformáció a karakterek, a tokenek és sorok számára hat, valamint az átlag és összeg típusú mértékekre a függvény esetén, és a modulra nézve is.

A bonyolult, vagy nem informatív változó rekord elnevezéseket javíthatja, csökkentheti a karakterek és a sorok számát, érthetőbbé, átláthatóbbá teszi a forrásszöveget és kommenteket tehet fölöslegessé.

A túl hosszú változó nevekkel viszont ronthat a karakterek és a sorok számán, az erlang design roule-ok [8] szabályainak betartását ellehetetleníti.

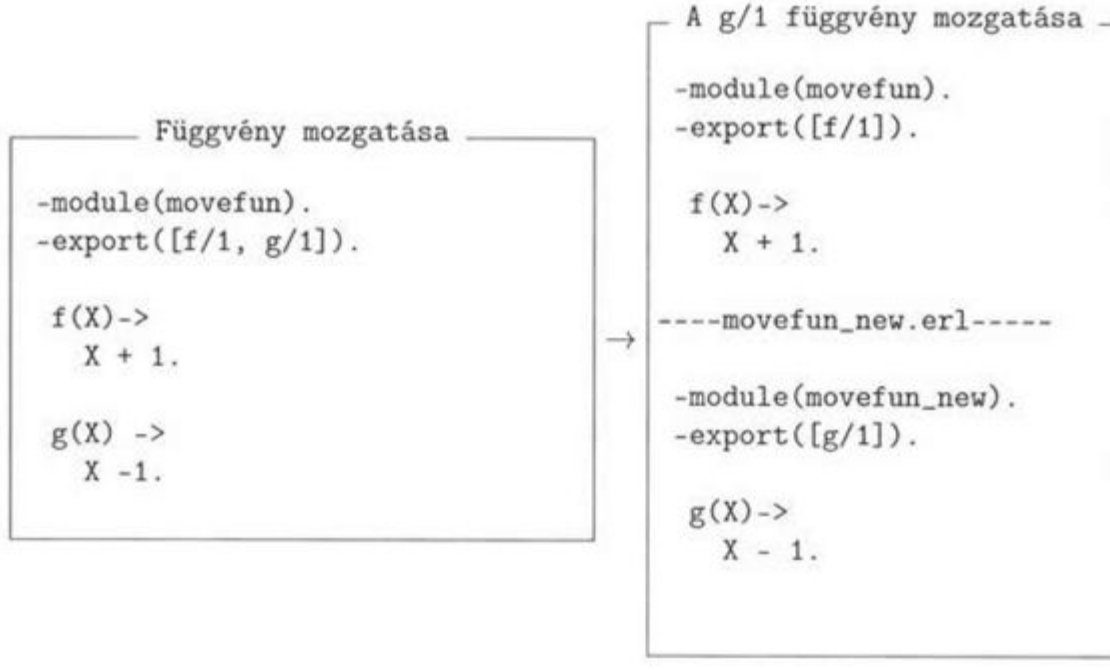
3.4.3. Átnevezések általában

A változó átnevezése, valamint a rekord és rekord mező átnevezése transzformációk hatást gyakorolnak a résztvevő függvényekre, azok moduljára, és fejléc fájlokkal láthatóvá tett rekordok és makrók alkalmazása esetén a láthatósági körbe eső modulokra.

Karakterek, tokenek és a sorok számára hatnak, valamint az átlag és összeg típusú mértékekre a függvényre és a modulokra nézve.

A túl hosszú változó nevekkel ronthatnak a karakterek és a sorok számán, az erlang design roule-ok [8] szabályainak betartását ellehetetlenítik.

3.4.4. Függvények modulok közötti mozgatása



3.2. ábra. Függvény mozgatása más modulba

Az átnevezések mellett a transzformációk következő nagyobb csoportja a programelemek mozgatását végző (*move* típusú) transzformációk. Ezek közül is a legérdekesebb a *move_function*, amely a kijelölt függvényeket egy másik modulba helyezi át. Természetesen ez a transzformációs lépés is (az előre és jól meghatározott szabályok betartásával) elvégzi a szükséges kompenzációs lépéseket, mint a kapcsolódó rekordok és makrók elérhetőségének a biztosítása, a függvényre irányuló és az onnan kiinduló minősített hívások kezelése, vagy cseréje. A transzformáció összetett, így a strukturális bonyolultsági mértékeket is markánsabban megváltoztatja, mint a *rename* típusú transzformációk, amelyeknél a programstruktúrát nem, csak a lexikális elemeket érinti a változás.

Másrészt a transzformációban résztvevő függvény csomópontok környezetét is nagyobb mértékben érinti a változás. Ha sorra vesszük azt, hogy pontosan mi változik, gyakorlatilag megkapjuk az erre a transzformációra vonatkozó szabály lépéseit.

- A függvényekben amelyeket transzformálunk (mozgatunk), a minősített hívásokon kívül más nem változik meg, de ezek a hivatkozások a *char_of_code*, a

line_of_code, a *max_length_of_lines* és a *McCabe* értékeinek a megváltozását eredményezik.

- A függvényt tartalmazó modulban természetesen ugyanezen okból, és mert a kompenzációs lépések a mozgatásokkal megszüntethetnek rekord, vagy makró definíciókat, az ezeket mérő két számmérték szintén megváltozik, tehát mindenképpen újra kell számolni őket.

  Ebben az esetben az érintett mértékek egyrészt a függvénnyel szoros kapcsolatban álló elemek darabszámai, a *number_of_fun*, a *number_of_records*, a *number_of_macros*. Megváltozik a funkcionális nyelvekre jellemző, és a függvényben szereplő elemek száma, mint a *number_of_funexpr*, ami a modulban szereplő függvény kifejezések számát adja vissza, az *otp_used*, amely az Erlang/OTP *behaviour* elemek számát adja, a *number_of_messpass*, vagyis az üzenetküldések száma, az *average_length_of_line*, és a látszat ellenére nagyon hasznos, a lexikális elemeket érintő *no_space_after_comma* mértékek.

- A forrás modulban az innen "elmozgatott" függvények megváltoztatják a *number_of_function*, vagyis a függvények száma nevű mértékeket, és a modul hívási gráfja (*call_graph*) is nagymértékben megváltozhat, ami szintén számos további mértéket befolyásolhat, úgy mint a *cohesion*, ami a függvények közötti hívások számán alapszik. Ugyanígy megváltozhat a befelé *function_calls_in*, valamint a kifelé irányuló hívások száma, a *function_calls_out* és a *number_of_funpath* - a modulban található függvény útvonalak száma.

- Mindezeken kívül nyilvánvalóan változik a *average_size*, vagyis az adott csomópontra mért átlagos érték modul és függvény esetén is.

Ahogy látjuk, gyakorlatilag az összes modulra vonatkozó mérték megváltozik, tehát ennél a transzformációs lépésnél, a hatékonyság növelése érdekében az érintett csomópontok számának a pontos meghatározásával, vagyis azok számának a lehető legkisebbre szorításával tudunk javítani.

Az érintett csomópontok a mozgatott függvények, és az azokat tartalmazó modulok, valamint azok a modulok, amelyekbe mozgatjuk a kijelölt függvényeket. Ezen csomópontokat érinti a leginkább a változás, tehát bele kell raknunk őket a listába, és újra kell számolni rajtuk a felsorolt bonyolultsági mértékeket.

3.2. megjegyzés. *Ezen a ponton kicsit módosítanunk kell a korábban ismertetett, az újraszámításhoz használt szabályrendszer protokollján, mert a korábbi feltételezésünkkel ellentétben nem elég csak a csomópontokat, a transzformációs lépést és a transzformációban résztvevő csomópontokat eltárolnunk. Szükség lesz arra az információra is,*

hogy melyik csomóponton milyen mértéket kell újra kiszámolnunk. Két csoportba oszthatjuk a csomópontokat. Ez egyik csoport tartalmazza a transzformációban résztvevő függvényeket és modulokat, a másik pedig a transzformációban érintett, vagyis a kapcsolódó csomópontokat. A protokollt az elemző skálázhatósága érdekében később még finomítanunk kell azzal, hogy az érintett csomópontokat megkülönböztetjük a típusuk alapján, vagyis más mértékeket rendelünk a függvényekhez, és másokat a modulokhoz.

A minősített hívást tartalmazó függvényekben csak az egyszerű számmértékek változnak meg, ahogy az őket implementáló modulokban is, tehát azokkal a mértékekkel, amelyeket a mozgatott és a mozgatás célját képező függvényben, valamint azok moduljában mérünk. A forrás modult importáló modulokban az importált függvény modulneve változik.

A transzformációban résztvevő programelemek: a mozgatott függvények, a mozgatott függvényeket eredetileg tartalmazó forrás modul, az átmozgatás célját képező modul, a mozgatott függvényeket hívó más függvények és azok moduljai, valamint az esetleg használt rekordok láthatóságának érdekében bevezetett fejléc fájlok.

Az újraszámításra kerülő strukturális bonyolultsági mértékek listája tehát egy mozgatott függvényre nézve a következő:

$(line\_of\_code, char\_of\_code, function\_sum, mcCabe,$
$max\_length\_of\_line, average\_length\_of\_line, no\_space\_after\_comma)$

A transzformált függvényeket tartalmazó forrás modul újraszámításra kerülő bonyolultsági mértékei:

$(line\_of\_code, char\_of\_code, number\_of\_fun, number\_of\_macros,$
$number\_of\_records, included\_files, imported\_modules, number\_of\_funpath,$
$function\_calls\_in, function\_calls\_out, cohesion, otp\_used, mcCabe,$
$number\_of\_funexpr, number\_of\_messpass, average\_size, coupling,$
$max\_length\_of\_line, average\_length\_of\_line, no\_space\_after\_comma,$
$max\_application\_depth, max\_depth\_of\_calling, min\_depth\_of\_calling,$
$max\_depth\_of\_cases, max\_depth\_of\_structs, number\_of\_funclauses)$

Minősített hívások esetén a transzformációban érintett függvények, valamint ebben az esetben azok modul csomópontjaira nézve is azonos a lista:

$cahr\_of\_code, line\_of\_code, max\_length\_of\_lines, no\_space\_after\_comma$

3.4.5. Elemzés összefoglalása

A transzformáció hatást gyakorol a résztvevő függvényre, de csak abban az esetben, ha az hív más függvényeket, vagy minősített függvényhívásokat tartalmaz. Hat a függvény moduljára, a hívásokon keresztül kapcsolódó függvényekre és modulokra, és természetesen a mozgatás céljaként kijelölt modulra.

A függvényekre nézve a sorok, a karakterek és a tokenek számát meghatározó, valamint az átlag/összeg típusú mértékére gyakorol hatást.

Megváltozik a modulok közti kapcsolatokat, a kimenő és bejövő függvényhívások száma, a belső kohézió és a modulok közti kapcsolatokat mérő értékek száma.

A függvények számának csökkentésével és növelésével a modulok méretét optimalizálja, a klaszterezési problémák megoldásához elengedhetetlen, és a függvények funkcionalitás szerinti gyűjtéséhez is hatékony. Javíthatja a *coupling* és a kohézió értékét.

A mozgatott függvények darabszáma túl nagyra növekedhet egy modulban. A makrókat és rekordokat használó függvények a definíciók mozgatásával (kompenzációk) túl bonyolult, vagy fölösleges kapcsolatrendszert hozhatnak létre a modulok között. A beágyazottsági mértékek, és a *McCabe* szám túlságosan megnőhet a modulokban. Ronthatja a kohéziót, a *coupling*-ot, a függvény kapcsolatok számát növelheti, valamint a kimenő és bejövő hívások számát rossz irányba változtathatja.

3.4.6. Rekord és makró mozgatása más modulba

Rekord és makró mozgatása

```
-module(mr02).
-record(person,
        {name, age}).
-record(name,
        {first, last}).

f() ->
  #person{name =
   #name{first="John",
   last="Do"},age=37}.
```

→

Rekord és makró

```
-module(mr02).
-include("person.hrl").

 f()->
   #person{name =
    #name{first="John",
    last="Do"}, age=37}.

----person.hrl----------

 -record(name,
     {first, last}).
 -record(person,
        {name, age}).
```

3.3. ábra. Rekord és makró mozgatása más modulba

A függvények mozgatásához hasonlóan a *rename_record*, és a makrókat mozgató *rename_macro* transzformációk is hatást gyakorolnak a bonyolultsági mértékekre saját és a mozgatás célját képező modulokban egyaránt.

A forrás modulban megváltoznak az alapvető számmértékek, mivel maga a rekord, vagy a makró eltűnik onnan és a cél modulban is, mivel ott megjelennek a mozgatást követően. Az eredeti modulban csökkennek az értékek, akkor a cél modulban megnőnek, így a *char_of_code*, *line_of_code*, *max_length_of_lines*, *McCabe*, *no_space_after_comma* mértékeket újra ki kell számolni.

A modulok függvényeit a változás ebben az esetben nem érinti. Előfordulhat az az eset is, amikor a transzformáció a rekordok láthatósága érdekében egy új fejléc fájlt készít, és ebben helyezi el a több fájl által használt makrókat, vagy rekordokat, és láthatóvá teszi ezeket minden érintett modulban. Ez a tény a bonyolultság szempontjából nem lényeges, de a transzformáció elemzése során mindenképpen említést érdemel.

Ebben az esetben a kapcsolódó modulokról, vagy függvényekről csak akkor beszélhetünk, ha valamelyik függvény használja az adott rekordot és annak láthatósága érdekében a moduljaik közös fejléc fájlra hivatkoznak. Ilyenkor a kapcsolódó modulokban a *rename_function* transzformációhoz hasonlóan az alapvető számmértékeket kell újra kiszámolni.

A transzformációban résztvevő programelemek: az átmozgatott makrókat, vagy rekordokat eredetileg tartalmazó forrás modul,az átmozgatás célját képező modul, a mozgatott makrókat, vagy rekordokat használó függvények moduljai fejlécfájlok használata esetén.

A mozgatott rekord, vagy makró moduljára nézve az újraszámításra kerülő bonyolultsági mértékek listája:

$$(line\_of\_code, char\_of\_code, number\_of\_macros, number\_of\_records, included\_files, imported\_modules, average\_size, max\_length\_of\_line, average\_length\_of\_line, no\_space\_after\_comma)$$

Minősített hívások létrehozása esetén a transzformációban érintett modul csomópontokat tekintve a lista:

$$(cahr\_of\_code, line\_of\_code, max\_length\_of\_lines, no\_space\_after\_comma)$$

3.5. Elemzés összefoglalása

A két transzformáció hat a tartalmazó modulra, minősített rekord, vagy makró alkalmazást tartalmazó függvényekre, és azok moduljaira, valamint a cél modul és a rekord, vagy makró láthatósági körébe eső modulokra.

A forrás és a cél modulban a sorok, karakterek, makrók, rekordok, importok és exportok száma változhat. A kacsolódó függvényekben és moduljaikban a sorok és karakterek száma a minősített hívások miatt.

A fölöslegesen rekord definíciók láthatóvá tétele miatt linkelt modulok fölösleges kapcsolatait szüntetheti meg, vagy függvények mozgatásánál a rekord definíciókat a megfelelő (felhasználási) helyre mozgatja.

3.5.1. Függvény paraméterek sorrendjének a cseréje

Paraméterek sorrendje

```
-module(reorder).
-export([sum/2]).

sum(A, B)
    when is_list(A)->
    A ++ [B].

sum(A, B) ->
    A + B.
```

→

A sum/2 paraméterei

```
-module(reorder).
-export([sum/2]).

sum(B, A)
    when is_list(A)->
    A ++ [B].

sum(B, A) ->
    A + B.
```

3.4. ábra. A *sum2* paramétereinek a cseréje

A *reorder_function_parameters* transzformáció rendkívül hasznos, mivel segítségével a függvény paraméterek csoportosítása, valamint rekordokba rendezése könnyedén megoldható. A bonyolultsági mértékek újraszámítása szempontjából nem bír nagy jelentőséggel, mivel egyetlen mérhető mértéket sem változtat meg komolyabban.

A transzformációhoz az egyetlen szabály, vagyis a szabály egyetlen lépése az, hogy *reorder* esetén nem kell semmilyen lépést végrehajtani. Beleértjük ebbe azt is, hogy nem kell megkeresni a függvényt, a modult, vagy a függvényhez közvetetten kapcsolódó modulokat (amennyiben függvény belsejében történt a változás), és a csomópont környezetével sem kell foglalkoznunk. Az, hogy nem végzünk el egyetlen lépést sem, csupán a hatékonyság növelése érdekében történik így. A mérések kimenetét tekintve az sem okozna problémát, ha az újraszámolásig minden lépésen végighaladnánk.

Ahogy azt korábban már jeleztük, ez a transzformáció is változtathat a forrásszövegen, ha nagyon hosszú azonosítókat használunk és a csere során két hosszú azonosító

egymás mellé kerül. Ilyenkor a *Pretty Printer* töri a sort, így megváltozhat a sorok száma, vagyis a *line_of_code* értéke.

3.5.2. Elemzés összefoglalása

A transzformáció hatással van a függvényre és annak moduljára. A függvényt hívó más függvényekben a csere a mértékekre nem gyakorolhat hatást, csak abban az esetben, ha a hosszú változó nevek cseréje sortörést eredményez.

Karakterek, tokenek és a sorok számára hat, valamint az átlag és összeg típusú mértékekre, a függvényre és a modulra nézve is.

Használata az *introduce_record*, *introduce_tuple*, és *inline* típusú transzformációkat készíti elő.

Szélsőséges esetben a sorok számát növelheti, ami így túllépheti az Erlang nyelvhez megfelelőnek ítélt, meghatározott értékeket [8].

3.5.3. Kifejezés kiemelése

Kifejezés kiemelése

```
-module(ef1).
-export([gcd/2]).

gcd(A,A) -> A;
gcd(A,B) ->
  {X,Y} = if
            A > B -> {A,B};
            true  -> {B,A}
          end,
  gcd(X-Y,Y).
```

→

A sort/2 létrehozása

```
-module(ef1).
-export([gcd/2]).

sort(A,B) ->
  if
     A > B -> {A,B};
     true  -> {B,A}
  end.

gcd(A,A) -> A;
gcd(A,B) ->
  {X,Y} = sort(A,B),
  gcd(X-Y,Y).
```

3.5. ábra. Kifejezés kiemelése a `sort/2` függvénybe a `gcd/2`-ből.

Az *introduce_function* (más nevén *extract_function*) transzformáció tetszőleges függvény egy kijelölt kifejezését (kizárólag teljes kifejezést) kiemeli a függvény törzséből, majd a kiemelt részből egy új függvényt állít elő, valamint elhelyez egy hívást az új függvényre az eredeti kifejezés helyén.

A transzformáció kezeli a kifejezésben használt változókat, és a kifejezésben kötött változók esetén tiltja a transzformációt, valamint a kifejezésen kívül kötött változókat elérhetővé teszi úgy, hogy ezek a változók formális paraméterként szerepelnek az

új függvényben. A transzformáció lokális a modulra, növeli a modul függvényeinek a számát, változtat a belső kohézióján, és megemeli a függvények hívási útvonalainak a számát, viszont nem gyakorol semmilyen hatást más modulokra és azok függvényeire. Az újraszámolási folyamatot ez a lokális hatás nagyon leegyszerűsíti, mert csak a függvényt tartalmazó modullal kell foglalkozni, és abban is csak az említett bonyolultsági mértékekkel. Az újonnan bevezetett függvényre viszont az összes bonyolultsági mértéket ki kell számolni és tárolni.

Mindezek mellett a kiemelést sok esetben azért végezzük el, hogy a vezérlő szerkezetek beágyazottsági mélysége kisebb legyen. Ez a változás, vagyis a beágyazottság mélységének a csökkenése megváltoztatja a *max_depth_of_cases* mértéket.

Az újraszámításra kerülő strukturális bonyolultsági mértékek listája az új függvény esetén az összes függvényeken értelmezhető bonyolultsági mérték.

A transzformációban résztvevő programelemek: a kiemelt kifejezést eredetileg tartalmazó függvény, az újonnan létrehozott függvény, és a függvényeket tartalmazó modul. A transzformáció tárgyát képző forrás függvényre ki kell számolni az következő mértékeket:

$$(cahr\_of\_code, line\_of\_code, max\_length\_of\_lines, max\_length\_of\_line,\\ average\_length\_of\_line, no\_space\_after\_comma, max\_application\_depth,\\ min\_depth\_of\_calling, max\_depth\_of\_cases, max\_depth\_of\_structs)$$

A transzformációban érintett modul csomópontra nézve az újraszámításra váró mértékek listája:

$$(char\_of\_code, number\_of\_fun, number\_of\_funpath, cohesion,\\ average\_size, max\_length\_of\_line, average\_length\_of\_line,\\ no\_space\_after\_comma, max\_application\_depth, min\_depth\_of\_calling,\\ max\_depth\_of\_cases, max\_depth\_of\_structs, number\_of\_funclauses)$$

3.5.4. Elemzés összefoglalása

A transzformáció hatással van a résztvevő függvényre, és annak moduljára. Használatát követően a függvények, sorok, karakterek száma megnövekedhet a modulban, de csökken a függvényben. A transzformáció lokális a modulra, de a függvény és a modul struktúrájára hatást gyakorol.

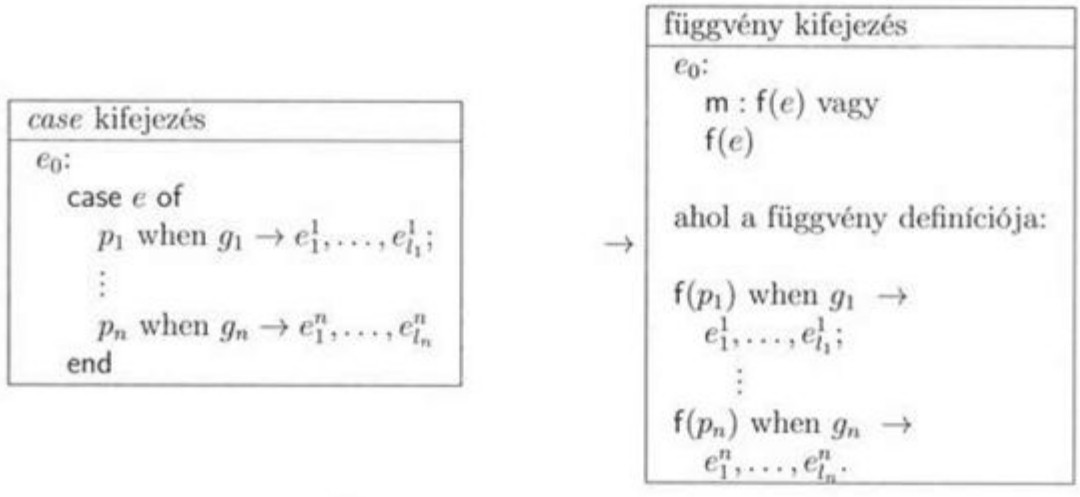
A beágyazottsági mértékekre jótékonyan hat. Alkalmazásával *McCabe* szám, és a beágyazottsági mértékek tekintetében jó eredményeket érhetünk el (lásd: 4. fejezetben). A transzformáció hatása tovább javítható a *move_function* transzformációval úgy, hogy a kiemelt függvényekből *library* modult készítünk. Mivel ezekre a függvényekre nincs

más függvényekből, vagy modulokból hivatkozás, semmilyen problémát nem okozunk az elmozgatásukkal, hacsak nem a *coupling* mérték csökkentése a célunk.

A transzformáció a függvények számát nagyon megnövelheti, és ezzel a *McCabe* számot is torzíthatja, mert minden újonnan behozott (kiemelt) függvény egyet hozzáad. Mindezeken kívül a sorok és karakterek számának növekedését is magával hozhatja.

3.5.5. *Case* kifejezés függvénnyé alakítása

A kifejezés kiemelése (*extract_case_expression*) a kifejezés kiemelése (*extract_function*) transzformáció speciális változata. A *RefactorErl*-ben futtatható transzformációk között nem is találjuk meg, kizárólag a hibadetektálást, és a forráskód optimalizálást végző algoritmus (bővebben a 4. fejezetben) használata során érhető el. A transzformáció működését tekintve annyival több a kifejezés kiemelésénél, hogy az átalakításra kijelölt kifejezést nem csak kiemeli egy új függvénybe, hanem át is alakítja azt a 3.6, és a 3.7 példákban látható módon.



3.6. ábra. Case kifejezés kiemelése

Az átalakítás kizárólag a *case* kifejezések kibontására alkalmazható. Úgy működik, hogy a kifejezés ágait függvény ágakká alakítja. A kifejezés eredeti helyén helyettesíti azt az új függvény hívásával és az új függvényben a *case* ágaival megegyező számú függvény klózban megtartja az ott eredetileg szereplő mintákat ($p_1, ..., p_n$), valamint az őr feltételeket ($g_1, ..., g_n$).

A transzformáció bonyolultságra gyakorolt hatása hasonló a kifejezés kiemelése transzformációnál elmondottakkal annyi különbséggel, hogy a *case* kifejezések számát, és ezzel párhuzamosan a beágyazottsági mélységüket is csökkenti. A hatása lokális a transzformált függvény moduljára, mivel az új függvényt nem exportálja, és a modul más függvényeit sem érinti a kifejezés kiemelésénél (*extract_function*) nagyobb mértékben. Az új függvényre itt is minden bonyolultsági mértéket ki kell számolni.

Case kifejezés kiemelése

```
-module(caset).
-export([f/1])

f(A) ->
  case A of
     {F, D} when
        is_integer(D) -> F(D);
          _ -> A
  end
```

→

A g/1 létrehozása

```
-module(caset).
-export([f/1])

f(A) ->
  g(A);

g({F, D})
   when is_integer(D) ->
  F(D);
g(A) ->
  A.
```

3.7. ábra. Kifejezés kiemelése a `sort/2` függvénybe a `gcd/2`-ből.

A transzformációban résztvevő programelemek: a kiemelt kifejezést eredetileg tartalmazó függvény, az újonnan létrehozott függvény, és a függvényeket tartalmazó modul. A transzformáció tárgyát képző forrás függvényre ki kell számolni a következő mértékeket:

$(cahr\_of\_code, line\_of\_code, max\_length\_of\_lines,$
$no\_space\_after\_comma, max\_depth\_of\_cases, max\_depth\_of\_structs)$

A transzformációban érintett modul csomópontokra nézve az újraszámításra váró mértékek listája:

$(line\_of\_code, char\_of\_code, number\_of\_fun, number\_of\_funpath, cohesion,$
$average\_size, coupling, max\_length\_of\_line, average\_length\_of\_line,$
$no\_space\_after\_comma, max\_application\_depth, min\_depth\_of\_calling,$
$max\_depth\_of\_cases, max\_depth\_of\_structs, number\_of\_funclauses)$

A transzformáció modulok és függvények bonyolultságára gyakorolt hatásai miatt a *extract_function* transzformáció mellett a 4. fejezetben részletesen foglalkozunk vele.

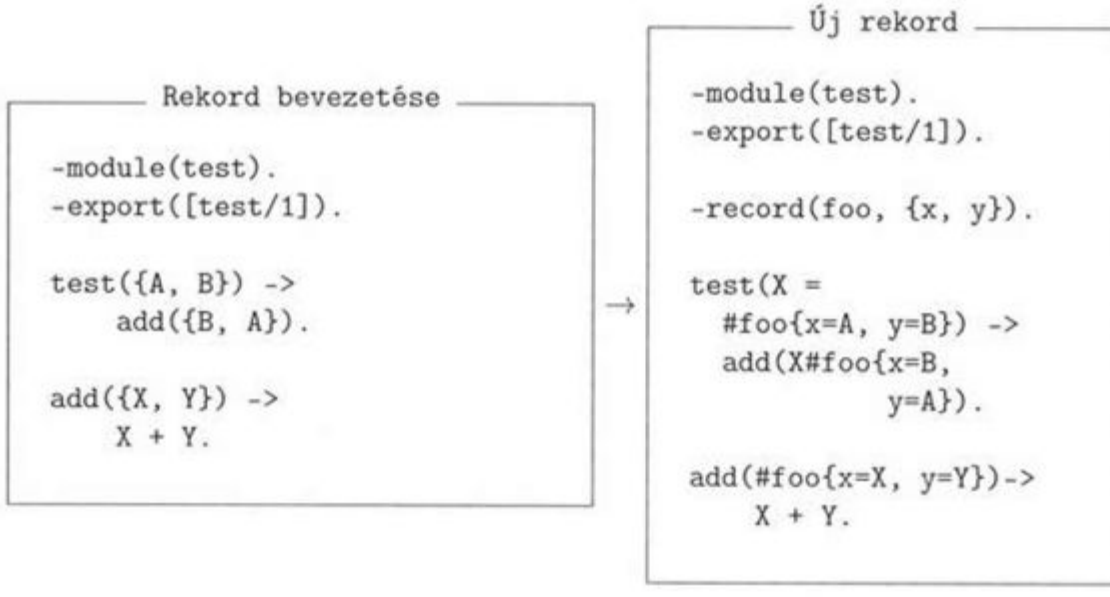
3.5.6. Elemzés összefoglalása

A transzformáció hatással van a résztvevő függvényre, és annak moduljára. A függvények, a sorok és a karakterek száma megnövekedhet a modulban, de ezzel együtt csökken a függvényben. A transzformáció lokális a modulra, de a függvény és a modul struktúrájára hatást gyakorol.

A beágyazottsági mértékekre jótékonyan hat. Alkalmazása mellett a *McCabe* szám, és a beágyazottsági mértékek tekintetében jó eredményeket érhetünk el.

A kiemelés a függvények számát szintén megnövelheti, így a *McCabe* számot torzíthatja.

3.5.7. Rekord bevezetése



3.8. ábra. A *foo* rekord bevezetése a *test* modulban

Az *introduce_record* transzformáció összetett és bonyolult kompenzációs lépéseket igényel. A transzformáció egy tetszőleges, kijelölt rendezett n-est rekorddá alakít úgy, hogy a forrás modulban létrehozza a megfelelő rekord definíciót. A kijelölt rendezett N-est átírja rekord kifejezéssé, amely hasonló formát kap, mint az eredeti n-es kifejezés:

$$\{e_1, e_2, \ldots, e_n\}$$

A létrejövő rekord nevet kap (a példában a *rec* nevet), a mezőkben pedig az eredeti formából kapott elemeket kötjük

$$\texttt{\#rec}\{f_1 = e_1, f_2 = e_2, \ldots, f_n = e_n\}$$

A rekord létrehozás során tehát a k^{th} elemet az $|\text{expr}\#\text{rec}.|f_k$ formára cseréljük le (a transzformáció kezeli a kifejezéseket is, de ez a bonyolultsági mértékek újraszámításához lényegtelen információ), valamint létrehozzuk a rekord definícióját a modulban, vagy azokban a modulokban, ahol erre szükség van.

A transzformációs lépés szükségessé teszi a kompenzációt, mivel elhagyásával az átalakított paraméterlista miatt a függvény hívási helyein hibát okozhatna. A forráskód helyességének megőrzése érdekében a kompenzáció két λ (lambda) függvény formájában realizálódik. A rekorddá konvertáláshoz amennyiben a rekordnév „*rec*" és a mezőnevek „f_1", „f_2", ..., „f_n" a kompenzációs függvény:

```
fun({V_1, V_2, ..., V_n}) ->
    #rec{f_1 = V_1, f_2 = V_2, ..., f_n = V_n}
end
```

A visszafelé alakításhoz, vagyis amikor rekordból n-esre konvertálunk a függvény a következő:

```
fun(#rec{f_1 = V_1, f_2 = V_2, ..., f_n = V_n}) ->
    {V_1, V_2, ..., V_n}
end
```

A transzformáció, ahogy láthatjuk, igen összetett, és ez a strukturális bonyolultsági mértékek változására is jelentős mértékben hat. Az átalakítást tartalmazó függvény nagymértékben megváltozik, ezért szinte minden bonyolultsági mértékét újra ki kell számítani.

A függvényt tartalmazó modul kap egy új rekordot, valamint megkapja az említett λ függvényeket tartalmazó kompenzációs makrókat, így az alapvető szám mértékeken kívül megnő a *number_of_macros*, a *number_of_records* értéke, és a függvény alkalmazások miatt a függvény útvonalak száma (*number_of_funpath*) is. Adatfolyam analízissel, valamint a hívási gráf alapján felderíthető kapcsolódó függvények, és azok moduljai a kompenzációk miatt szintén megváltoznak, de ezek csak az alapvető számmértékek mentén.

A transzformációban résztvevő programelemek: a transzformált függvény, vagyis amelynek a paramétereit, vagy a függvény blokkban található rendezett n-est átalakítjuk rekorddá, a transzformált függvény modulja, amely megkapja rekordot és a kompenzációkat, a rekord láthatóságának érdekében bevezetett fejléc fájl, a hívási gráf alapján a transzformált függvényt hívó más függvények, és az azokat tartalmazó modulok. Az újraszámításra kerülő strukturális bonyolultsági mértékek listája a transzformált függvényre nézve:

(*line_of_code*, *char_of_code*, *function_sum*, *max_depth_of_calling*, *min_depth_of_calling*, *number_of_funexpr*, *max_length_of_line*, *average_length_of_line*, *no_space_after_comma*)

A transzformációban résztvevő függvény moduljára alkalmazandó mértékek listája:

$(line\_of\_code, char\_of\_code, number\_of\_fun, number\_of\_macros,$
$number\_of\_records, included\_files, number\_of\_funpath,$
$mcCabe, number\_of\_funexpr, average\_size, max\_length\_of\_line,$
$average\_length\_of\_line, no\_space\_after\_comma)$

A transzformált csomóponthoz kapcsolódó, vagyis a transzformációban közvetetten érintett függvények listája:

$(line\_of\_code, char\_of\_code, function\_sum, max\_depth\_of\_calling,$
$min\_depth\_of\_calling, number\_of\_funexpr, max\_length\_of\_line,$
$average\_length\_of\_line, no\_space\_after\_comma)$

A transzformált függvényhez kapcsolódó más függvények moduljaira vonatkozó lista:

$(line\_of\_code, char\_of\_code, number\_of\_fun, number\_of\_macros,$
$number\_of\_records, included\_files, number\_of\_funpath, number\_of\_funexpr,$
$average\_size, max\_length\_of\_line, average\_length\_of\_line,$
$no\_space\_after\_comma)$

3.5.8. Elemzés összefoglalása

A transzformáció hat a résztvevő függvényre, annak moduljára, a hívással kapcsolódó függvényekre, valamint ezek moduljaira.

Sorok és karakterek számát növelheti, de csak kis mértékben. A rekord paraméterrel ellátott függvényeket hívó más függvények megváltoznak, de ezek is kis mértékben. A sorok és karakterek számában viszont a kompenzációs makrók bevezetése miatt a modulok nagymértékben változnak.

A sorok, karakterek, rekordok és makrók száma, valamint a bevezetett fejléc fájlok nagyon sok mértéket megváltoztatnak. A beágyazottsági mélységek nem változnak és a *McCabe* szám sem. A kohézióra és *coupling*-ra nem hat.

A kompenzációs makrók növelik a sorok és karakterek számát, és a rosszul megválasztott, vagy generált rekord és mezőnevek olvashatatlanná tehetik a programszöveget.

3.5.9. Kifejezés változóhoz kötése

Kifejezés kötése változóba

```
-module(merge).
-export([f/1]).

f(X) ->
  A = (X + 1),
  g(X + 1),
  [X + 1 || X <- {1, 2}].

g(A) ->
  A.
```

→

Az x+1 kifejezés kötése

```
-module(merge).
-export([f/1]).

f(X) ->
  H = X + 1,
  A = (H),
  g(H),
  [X + 1 ||
    X <- {1, 2}].

g(A) ->
  A.
```

3.9. ábra. Az *x+1* kifejezés kötése az *A* változóba

A kifejezés kötése, más néven az *introduce_variable* a többször előforduló, de azonos kifejezések kiküszöbölésére használható transzformáció. Működését tekintve összetett és meglehetősen bonyolult lépés, de a strukturális bonyolultsági mértékeket tekintve lokális a függvényre. Nincs hatással a függvényt hívó további függvényekre és azok moduljaira sem, vagyis ebből a szempontból a hatása azonos a *rename_variable* lépésnél használatosakkal. Természetesen a kifejezést tartalmazó függvény és a modulja megváltozik, így mindkettőnél újra kell számolni néhány bonyolultsági mérőszámot.

A transzformáció a kijelölt kifejezést egy megadott nevű változóhoz köti, majd az eredeti kifejezés összes előfordulását lecseréli egy új, a lépésben bevezetett változóra, megnövelve, vagy éppen csökkentve a karakterek, a sorok és a változók darabszámait.

A transzformációban résztvevő programelemek: a transzformációban résztvevő függvény, és a transzformációban résztvevő függvény modulja. Az újraszámításra kerülő strukturális bonyolultsági mértékek listája a transzformációban résztvevő függvényre nézve a következő:

$(line\_of\_code, char\_of\_code, function\_sum, max\_length\_of\_line,$
$average\_length\_of\_line, no\_space\_after\_comma)$

A transzformáció tárgyát képző függvény moduljára alkalmazandó mértékek:

$(line\_of\_code, char\_of\_code, average\_size, max\_length\_of\_line,$
$average\_length\_of\_line, no\_space\_after\_comma)$

3.5.10. Elemzés összefoglalása

A transzformáció hat a változót/kifejezést tartalmazó függvényre, valamint annak moduljára.

A sorok, karakterek és a tokenek számára van hatással, de a struktúrát nem változtatja meg. A transzformáció hatása lokális a függvényre és a modulra. A sorok és karakterek számát megnövelheti, de csökkentheti a transzformációban megadott aktuális paraméterektől függően. A sorok számának és hosszának növekedésével átlépheti az Erlang nyelvhez elfogadott normákat.

3.5.11. Paraméterek összevonása rendezett n-esre

Paraméterek összevonása

```
-module(tuplefunpar).
-export([sum/2]).

sum(A, B) ->
  A + B.
```

→

A sort/2 paraméterei

```
-module(tuplefunpar).
-export([sum/1]).

sum({A, B}) ->
  A + B.
```

3.10. ábra. A *sort/2* paramétereinek összevonás rendezett n-essé

Az *introduce_tuple* transzfromációs lépés egy tetszőlegesen kijelölt függvény paramétereit rendezett n-essé alakítja, vagyis összevonja a függvény kijelölt paramétereit, és *{}* zárójelek közé helyezi azokat. Az összevonás nem érinti komolyan a bonyolultsági mértékeket, de a hatása globális, vagyis szétterjed a függvény környezetére is. Érinti a függvény modulját, valamint a függvényt hívó további függvényeket és moduljaikat.

A transzformációban résztvevő programelemek: a transzformációban résztvevő függvény, a transzformációban résztvevő függvény modulja, a transzformációban érintett függvények, és ezek moduljai.

Az újraszámításra kerülő strukturális bonyolultsági mértékek listája a résztvevő függvényre nézve a következő:

$(line\_of\_code, char\_of\_code, otp\_used, function\_sum, max\_length\_of\_line,$
$average\_length\_of\_line, no\_space\_after\_comma)$

A transzformáció tárgyát képző függvény moduljára alkalmazandó mértékek listája:

$(line\_of\_code, char\_of\_code, average\_size, max\_length\_of\_line,$
$average\_length\_of\_line, no\_space\_after\_comma)$

A transzformációban érintett függvényekre, és azok moduljaira a lista azonos, mivel a hívási helyeken a függvény aktuális paraméterlistája hasonlóképp bővül, mint a formális paraméterlista a függvény definíciójánál.

3.5.12. Elemzés összefoglalása

A transzformáció hatással van a résztvevő függvényre, annak moduljára, a kapcsolódó függvényekre, és azok moduljaira.

Hatással van a sorok és a karakterek számára, a kompenzációs makrók miatt a kapcsolódó és résztvevő modulokra is. A résztvevő függvény hívási helyén is változtat ugyanezen értékeken. A függvények hívási helyein is megnő a sorok és karakterek száma, valamint a kompenzációs makrók, és a láthatóságukat biztosító fejléc fájlok a modulok kapcsolatait megváltoztatják. Az importált modulok számai is megváltozhatnak.

Az *introduce_record* lépést készítheti elő azáltal, hogy a valamilyen szempontból egybe tartozó adatokat összerendeli, így a rekorddá alakítás automatikusan elvégezhető ezekre az összerendelt a csoportokra.

A sok kompenzációs makró, és a fejléc fájlok bevezetése a modul kapcsolatok számát növelheti, valamint a sorok és karakterek száma átlépheti az elvárt értékeket.

3.5.13. Függvény hívás eliminálása

Függvény hívás eliminálása

```
-module(inlinefun).
-export([sort/2]).

sort(A, B) ->
   case is_greater(A, B) of
     true -> {A, B};
     false -> {B, A}
   end.

is_greater(A, B) ->
   A > B.
```

→

Az is_greater/2 függvény

```
-module(in5).
-export([sort/2]).

sort(A, B) ->
    case A > B of
      true -> {A, B};
      false -> {B, A}
    end.

is_greater(A, B) ->
   A > B.
```

3.11. ábra. Az *is_greater/2* függvény hívásának eliminálása

Az *eliminate_function_call* transzformáció feladata, hogy megszüntesse a kijelölt függvényhívást, majd helyettesítse azt a hívott függvény törzsével. A transzformáció az eredetileg meghívott függvényt nem változtatja meg és nem is törli akkor sem, ha arra nincs máshonnan hívás a modulon belülről, vagy kívülről. A transzformáció hatása lokális az átalakításban résztvevő, vagyis a transzformált függvényre nézve, de

a bonyolultsági mértékeket tekintve hatással van a függvényt tartalmazó modulra, valamint arra a függvényre is, amelyet helyettesítettünk. Az ezekre történő hívások (*calls_for_function*, *function_calls_in*, stb.) a modulon belülre, és kívülre nézve is megváltoznak, így a változtatások kis mértékben ugyan, de az érintett, függvényeket tartalmazó modulokra is hatással vannak.

A transzformációban résztvevő programelemek: a transzformációban résztvevő függvény, a függvény modulja, a transzformációban érintett függvények és moduljaik.

Az újraszámításra kerülő strukturális bonyolultsági mértékek listája a transzformált függvényre nézve hosszú, mivel az átalakítás helyéről eltűnik egy függvényhívás, amely befolyásolhatja a függvényhívási láncok mélységét, valamint behozhat a függvénybe számos olyan nyelvi konstrukciót, amik korábban nem szerepeltek. Ezeken kívül megváltozhatnak a különböző egymásba ágyazottságok mélységei azáltal, hogy a függvényhívást felváltó függvény törzse tartalmazhat *case*, *if*, üzenetküldő, és egyéb, a bonyolultsági mértékek által mért kifejezéseket.

$(line\_of\_code, char\_of\_code, max\_depth\_of\_calling, min\_depth\_of\_calling,$
$max\_length\_of\_line, average\_length\_of\_line, no\_space\_after\_comma,$
$branches\_of\_recursion, McCabe, calls\_for\_function, calls\_from\_function,$
$number\_of\_fuexpr, otp\_used, number\_of\_messpass, number\_of\_fuexprs)$

A résztvevő függvény moduljára alkalmazott mértékek listája nyilván itt is függ a transzformált függvényre alkalmazott listától abban az esetben, ha az adott mértéket modul, és függvény csomópontra is tudjuk mérni, mivel a függvényeit mérő mértékek megváltozása magával hozza a modul mérőszámainak változását is (a modul értékei számos esetben a függvények összesített értékeiből adódnak).

$(line\_of\_code, char\_of\_code, min\_depth\_of\_calling, max\_length\_of\_line,$
$no\_space\_after\_comma, number\_of\_fun, number\_of\_records,$
$module\_sum, included\_files, imported\_modules, number\_of\_funpath,$
$McCabe, max\_depth\_of\_cases, max\_application\_depth, max\_depth\_of\_structs)$

A transzformációban érintett függvényre vonatkozó lista, amelybe beletartozik a korábban hívott, de a transzformáció által lecserélt függvényhíváshoz tartozó függvény definíciója is. A lista rövid, és a függvényre vonatkozó hívásokat érinti.

$(calls\_for\_function, function\_calls\_in, function\_calls\_out)$

Az érintett modul csomópontokra nézve a lista szintén rövid, és csak abban az esetben van jelentősége, mikor a transzformációban lecserélt függvényhíváshoz tartozó

függvény más modulban van, valamint minősített, vagy importált hívással alkalmazták a transzformáció előtt. Ezekben az esetekben megváltozhat a modul összesített mérőszáma, a függvény útvonalak száma, és a függvénykapcsolatok struktúrája.

$(module\_sum, number\_of\_funpath, function\_calls\_in)$

3.5.14. Elemzés összefoglalása

A transzformáció hat a résztvevő függvényre, a függvény moduljára, és minősített hívások esetén a kapcsolódó modulokra.

A függvény hívások számát csökkenti, de a beágyazottsági mértékekre, a struktúrákra, és ezzel együtt a *McCabe* mértékre is hat. A sorok és a karakterek számát kis mértékben csökkentheti. Az export listák méretét csökkenti, így az exportált függvények számát is. A minősített hívások esetén más modulokra is hatással lehet.

Javíthatja a függvények számát, a *McCabe* értékét a modulra nézve. Csökkentheti a sorok és karakterek számát is.

Elronthatja a beágyazottsági mértékek értékeit. A *McCabe* számot a függvényekre nézve nagyon elronthatja, és a beágyazottsági mélységeket is rossz irányba változtathatja, ha nem kellő körültekintéssel alkalmazzuk.

3.5.15. Makró alkalmazás eliminálása

Az *eliminate_macro* transzformáció egy tetszőlegesen kiválasztott makró alkalmazását, vagyis a makróra való hivatkozást cseréli le azon az egy helyen, amit kijelöltek, és a többi hivatkozást változatlanul hagyja.

A művelet lokális a függvény törzsére. Amennyiben a hívás a függvény formális paraméter listájában van, a függvényt hívó függvények nem változnak, mivel számukra a makró törzsében elhelyezett programelemek láthatóak csak, a makró behelyettesítés miatt.

Más részről a makró definícióját sem érinti a változás, így a fejléc fájlban tárolt definíciók sem változnak meg, de a makróra való hivatkozások száma, vagyis a belső hívási útvonalak száma, valamint a modul kohéziója változik.

Képzeljük el azt az egyszerű esetet, mikor a makró törzsében kizárólag egy atomot helyezünk el *-define(Macro, macrobody)*, és egy függvény formális paraméter listájában hivatkozunk a makróra *f(?Macro) -> ....* Ez pontosan olyan, mintha egy atommal kellene paraméterezni a függvényt, mivel a makró helyettesítés során a benne található atomra hivatkoznunk *f(macrobody) -> ....*

A függvény meghívásakor a makróban található atommal kell azt paraméterezni *f(macrobody)...*, így a makró alkalmazás eliminálása nincs hatással a hívási helyekre, ezáltal nem érinti a hívó függvény és modulja bonyolultságát sem.

Makró hívás eliminálása

```
-module(inlmac).
-define(Macro(A),-A).

f() ->
  ?Macro(3),
  ?Macro(4).
```

→

Makró helyettesítése

```
-module(inlmac).
-define(Macro(A),-A).

f() ->
  -3,
  ?Macro(4).
```

3.12. ábra. Makró hívás eliminálása a *f/0* függvényben

Amennyiben itt is eliminálják a makrót, akkor a hívó függvény már a transzformáció tárgyát képezi, és az ott bemutatott szabályok érvényesek rá.

A transzformációban résztvevő programelemek: a transzformációban résztvevő függvény és modulja. Az újraszámításra kerülő strukturális bonyolultsági mértékek listája a résztvevő függvényre nézve a következő:

$(line\_of\_code, char\_of\_code, function\_sum,$
$max\_length\_of\_line, average\_length\_of\_line, no\_space\_after\_comma)$

A transzformáció tárgyát képező függvény moduljára alkalmazandó mértékek listája:

$(line\_of\_code, char\_of\_code, cohesion, average\_size,$
$max\_length\_of\_line, average\_length\_of\_line, no\_space\_after\_comma)$

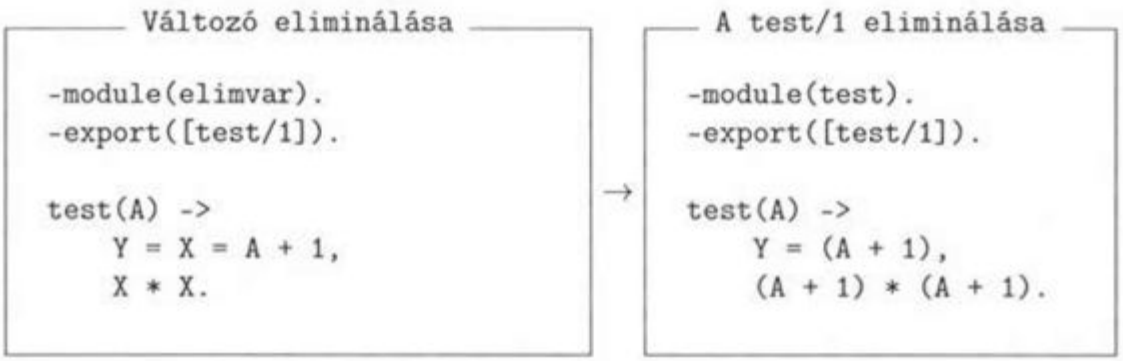
3.5.16. Elemzés összefoglalása

A transzformáció hat a résztvevő függvényre, annak moduljára, és fejléc fájlok alkalmazása esetén a kapcsolódó modulokra.

A függvényben és a modulban a karakterek és a sorok számát növelheti, vagy csökkentheti attól függően, hogy a makró neve, hosszabb, vagy rövidebb, mint a helyettesített értéke (az ebben szereplő karakterek száma). Amennyiben a makrót több modulban is használták, ezekre is, és a függvényeikre is hatással lehet, de ebben az esetben is csak a karakterek és a sorok számára van hatással, a strukturális mértékek értékeire nem.

A sorok és karakterek számát megnövelheti, de csökkentheti is alkalmazásának körülményeitől függően. A sorok számának és hosszának növekedésével átlépheti az *erlang design roul*-okban [8] definiált értékeket.

3.5.17. Változó eliminálása



3.13. ábra. A *test/2* függvény változójának az eliminálása

A változó eliminálása (*eliminate_variable*) transzformáció egy kijelölt változó összes előfordulását lecseréli a változóba kötött kifejezésre, és megszünteti a változó kötését tartalmazó kifejezést. A lépés lokális arra a függvényre, ahol végrehajtják, ezáltal a bonyolultságot is csak nagyon kis mértékben változtatja meg. (Ez a transzformáció ellentettje a kifejezés kötése változóban, más néven az *introduce_variable* transzformációnak.)

A transzformációban résztvevő programelemek: a transzformációban résztvevő függvény és modulja. Az újraszámításra kerülő strukturális bonyolultsági mértékek listája a résztvevő függvényre nézve a következő:

$(line\_of\_code, char\_of\_code, function\_sum,$
$max\_length\_of\_line, average\_length\_of\_line, no\_space\_after\_comma)$

A transzformáció tárgyát képző függvény moduljára kiszámolandó mértékek listája:

$line\_of\_code, char\_of\_code, cohesion, average\_size,$
$max\_length\_of\_line, average\_length\_of\_line, no\_space\_after\_comma$

3.5.18. Elemzés összefoglalása

A transzformáció hat a változót tartalmazó függvényre, valamint annak moduljára.

A sorok, karakterek és a tokenek számára hatást gyakorolhat, de a program struktúráját nem változtatja meg.

A sorok és karakterek számát megnövelheti, de csökkentheti is, és ebből következik, hogy alkalmazásával átléphetjük a design roule-okban [8] definiált, elvárt értékeket.

3.6. Kivételek és rendhagyó esetek

Ahogy azt a fejezet elején említettük, a *Refactoring* alapú, és a kód szerkesztéséből adódó transzformációk esetén a szemantikus gráf változásának három fő műveletét különböztethetjük meg, ezért minden, a forráskód megváltoztatását érintő lépés e három fő művelet valamelyikét kezdeményezi.

1. Az első eset az törlés (*remove*), ahol egy adott csomópontot törlünk a gráfból. A csomópont lehet *file*, *module*, *form*, *expression*, és még számos egyéb elem, hogy csak a legfontosabbakat említsük.

   Mivel a *RefactorErl*, és egyáltalán az Erlang fordítóprogramja a minősített hívások esetén nem vizsgálja, hogy egy hívás mögött van-e függvénydefiníció, erre a tényre az ellenőrző rendszernek is fel kell készülnie. Ez azt jelenti, hogy a törölt függvényt hívó egyéb függvények esetén a bonyolultsági mértékek megváltoznak akkor ha a hívott függvényt törölték és ez a fordítás (vagy a szemantikus gráf építése közben) nem jelentkezik hibaként.

   Amennyiben modulon belül található a függvényre való hivatkozás, és azt törlik, a fordítás ezt már hibaként értékeli, vagyis nem kell foglalkoznunk az esettel.

   A törlés során az aktuálisan törölt függvényt, vagy modul csomópontot természetesen nem kell újra ellenőrizni. Függvény törlésekor csak a modul mérőszámai változnak, és ahogy említettük, a minősített hívásokon keresztül kapcsolódó más függvények mérőszámai.

   Ezekből a példákból láthatjuk, hogy a különleges csomópontok a *függvény*, és a *modul* típusúak, ami nem is meglepő akkor, ha a strukturális bonyolultsági mérőszámokat is az ilyen típusú elemeken értelmezzük. A bonyolultság kiszámításához a modul és függvény csomópontok gyűjtése automatikusan kihagyja a törölt elemeket, ezért a probléma csak elméleti, és nem technikai jellegű.

2. A beszúrás (*insert*) művelet esetén a dolog éppen fordítva van, mint az átnevezés (*remove*) esemény bekövetkeztekor, mivel itt egy csomópontot szúrunk be a szemantikus gráfba. Ekkor is a *modul*, valamint a *függvény* csúcsokkal lehet probléma.

   Az új, éppen beszúrt függvényre nézve minden bonyolultsági mértéket ki kell számolni és tárolni, modul esetén pedig a modulra kell megtenni ugyanezt. A korábban említett minősített hívások miatt előállhat az az eset is, hogy korábban töröltünk egy függvényt, amely hívott más függvényeket, és ezt a függvény most újra létrehozzuk, vagyis beszúrjuk. Ilyenkor a kapcsolódó függvények, és moduljaik bonyolultsági mértékeit újra kell kalkulálni, de az automatikus keresés miatt

a gyakorlati megvalósításnál (implementáció) ennél a problémánál sem kell külön számolnunk az esettel.

3. A harmadik művelet a frissítés (*update*), amely esemény akkor következik be, ha valamely csúcsot megváltoztatjuk (az átnevezés (*update*) esemény is kezdeményezheti). Erre az esetre a fentebb bemutatott szabályrendszer megszorítások, és a különleges esetek vizsgálata nélkül alkalmazható.

Amennyiben a három közül egyszerre több is megtörténik, minden esetben meg kell nézni, hogy mely elemekre és mely bonyolultsági mértékek változnak, és ezeket össze kell fésülni, és csak ez után végrehajtani (alkalmazni).

3.7. Nem *refactoring* alapú transzformációk

A nem *refactoring* hatására történő, szemantikus gráfon eszközölt változások, vagyis a forrásszöveg bármilyen jellegű megváltoztatása, és mentése esetén is meg lehet különböztetni olyan eseteket, amelyek eltérő módon hatnak a bonyolultságra.

A változásokat ekkor nem a transzformáció típusa alapján tudjuk megkülönböztetni, hanem a transzformált programelem típusa alapján, de a környezetre gyakorolt hatások itt nehezen behatárolhatóak, mivel nehéz megmondani, hogy egy modul frissítésekor mi változott meg, és mi nem, és hogy a változásnak lokálisak-e a hatásai, vagy a teljes forrásszöveget érintik.

Ilyenkor a legjobb módszer az, ha megvizsgáljuk, hogy a transzformációban érintett függvény mely más modulokkal van kapcsolatban a függvény hívási gráfon keresztül, és ezekben a modulokban újraszámoljuk a strukturális bonyolultsági mértékeket (az összeset).

Ezt a lépést felfoghatjuk úgy, mint az "egyéb eset", és a programszöveg minden olyan mentésekor felkészülünk erre, amikor nem detektálható *refactoring* által kezdeményezett változás.

A probléma megoldása nem bonyolult, csak a hatékonysága rosszabb a "nevesített" transzformációkénál. Sajnos a legrosszabb esetben az összes bonyolultsági mértéket ki kell számolni minden modulra, és a modulokban található valamennyi függvényre, de a mérési folyamat ekkor is pontosan annyi időt vesz igénybe, mint a forrásszöveg első betöltésekor lefuttatott mérések elvégzése.

Az előzőekben ismertetett eljárás tehát alkalmas arra, hogy megtalálja azokat a kódrészleteket, amelyek megsértik a fejlesztő által definiált strukturális bonyolultsági mértékeket, vagyis amelyek eltérnek az adott programra alkalmazott (alkalmazható) standard mérőszámoktól.

Az viszont, hogy felderítjük az ilyen jellegű problémákat, önmagában sajnos nem elegendő a szoftverfejlesztés támogatására, mivel a felhasználónak nincs információja

arról, hogy a tárolt modul, és függvény csomópontok a forráskódban mely függvényekhez és modulokhoz tartoznak.

Ezt az információt is elérhetővé kell tennünk a számára, és ezt meg is tehetjük, ha az elemzés befejeztével megjelöljük az aktuálisan használt modulban az elemző által bonyolultnak vélt programrészeket, valamint listába gyűjtve megmutatjuk a forrásszöveg aktuálisan nem megnyitott, vagy nem látható részeiben található eltéréseket. (Az elemző implementált változata a 6.1 és a 6.2 ábrán látható).

3.8. Összefoglalás

A fejezetben megismerhettük azt a bonyolultsági mértékeken alapuló elemző algoritmust, amely segít az erlang programszövegekben alkalmazott, kezelhetetlenül bonyolult, vagy nem megfelelő nyelvi konstrukciók felderítésében, valamint a különböző fejlesztési problémát okozó programrészek összegyűjtésében.

Az algoritmus alapja a forrásszövegből épített szemantikus gráf (lásd: 2. fejezetben) vizsgálata (ami természetesen ebben az esetben is a strukturális bonyolultsági mértékek mérésén alapszik), de ezen a ponton már lehetőségünk van az alapértelmezett értékek definiálására, amely értékeket az elemző összehasonlít a forrásszövegen aktuálisan mért értékekkel, és az így megtalált eltéréseket jelezni tudja.

A fejezetben ismertetésre került az a szabályrendszer, amely alapján az elemző algoritmus eldönti, hogy a forrásszövegen eszközölt transzformációk során a szemantikus gráf, vagyis a programszöveg mely részei változtak meg az adott transzformáció hatására, és melyik programelemekhez milyen bonyolultsági mértékeket kell újra kiszámítani. A hatékony újraszámítási folyamatok kivitelezése érdekében csak az adott transzformációnál meghatározott mértékek, és csak a transzformáció által érintet gráf csomópontok kerülnek újbóli elemzésre.

A szabályrendszer kidolgozása és elemzése érdekében az egyes transzformációk különkülön szekciót kaptak. Ezek részletesen bemutatják az adott transzformáció működését és a szemantikus gráfra, vagyis a forrásszövegben elkülöníthető programnyelvi konstrukciókra gyakorolt (bonyolultságot érintő) hatásukat.

A transzformációk hatáselemzése elméleti úton, következtetések alapján történt, de az így tett állításokat az implementált elemző prototípus felhasználva, tesztek futtatásával és elemzésével sikerült igazolni.

3.9. A tézis megfogalmazása

Bonyolult programrészek lokalizálása, és kódjavítást célzó programtranszformációk hatása a kódminőségre.

Létrehoztuk azt az elemző algoritmust, amely a programszöveget jellemző bonyolultsági mértékek alapján a nem megfelelő programozási stílussal készült, vagy a kezelhetetlenül bonyolult programkonstrukciókat feltárja a szoftver teljes életciklusa során.

Kidolgoztuk továbbá azt a szabályrendszert, amely felhasználásával az elemző folyamatok minden mérés során a szemantikus gráfnak kizárólag azt a részgráfját mérik, amelyre a programszöveget megváltoztató transzformációs lépések hatással vannak.

3.10. Releváns publikációk

- Király, R., Kitlei R.: *Complexity measurments for functional code* 8th Joint Conference on Mathematics and Computer Science (MaCS 2010) refereed, and the proceedings will have ISBN classification July 14-17, 2010

- Király, R., Kitlei R.: *Implementing structural complexity metrics in Erlang.* '10 ICAI 2010 – 8th International Conference on Applied Informatics to be held in Eger, Hungary January 27-30, 2010

- Király, R. and Kitlei, R.: *Implementing structural complexity metrics for Erlang* Poster on the 8th International Conference on Applied Informatics, ICAI 2010, 2010

- László Lövei, Zoltán Horváth, Tamás Kozsik, and Roland Király. *Introducing records by refactoring.* In Proceedings of the 2007 ACM SIGPLAN Erlang Workshop, pages 18-28. ACM Press, 2007.

- László Lövei, Zoltán Horváth, Tamás Kozsik, Roland Király, and Róbert Kitlei. *Static rules of variable scoping in Erlang.* In Emőd Kovács, Péter Olajos, and Tibor Tómács, editors, Proceedings of the 7th International Conference on Applied Informatics, volume 2, pages 137-145, 2008. rev: Zbl pre05662517.

- Lövei, L., Horváth, Z., Kozsik, T., and Király, R.: *Static rules for variable scoping in Erlang* In The 7th International Conference on Applied Informatics, ICAI 2007, Eger, Hungary, January 2007

- István Bozó, Dániel Horpácsi, Zoltán Horváth, Judit Kőszegi, Roland Király, Róbert Kitlei, Máté Tejfel, Melinda Tóth,. *Haladó technológiák szoftverrendszerek forráskódú elemzésére A RefactorErl hatékonyságának és felhasználói felületének továbbfejlesztése*

  Készült az Ericsson Magyarország Kft megbízásából és támogatásával a KMOP-1.1.2-08/1-2008-0002 projekt keretében az ERFA támogatásával Tech report 2011 Ericsson Hungary

4. fejezet

Automatikus programtranszformációk

4.1. Bevezetés

A különböző programtranszformációs lépéseket követően megmérhető a forrásszöveget jellemző strukturális bonyolultsági mértékek változása. Ahogy azt korábban már láthattuk a 3. fejezetben, a bonyolult programrészeket lokalizáló algoritmus megjelöli azokat a nehezen kezelhető programrészeket is, amelyeket a fejlesztő nem biztos, hogy észrevesz, vagy a program nagy mérete miatt nem kerül a látókörébe. Mindezek ellenére a programkód "kézzel" történő átalakítása, javítása bonyolult és nehézkes feladat.

A probléma megoldásához vegyük ismét sorra, hogy milyen lehetőségeket kínál a 3. fejezetben korábban ismertetett elemző és hibadetektáló algoritmus.

- Használatával a 2. fejezetben definiált $\mathcal{SG}$ szemantikus gráfban tárolhatjuk a forrásszöveget, majd a gráf alapján vizsgálhatjuk (mérhetjük) a bonyolultságát, valamint a szintaktikai és szemantikai tulajdonságait.
- Biztosítja azokat a transzformációs lépéseket, amelyek a megfelelő kompenzációk elvégzése mellett a szemantikus gráf megváltoztatásával képesek átalakítani a forrásszöveget, majd az eredményt újból forrásszöveggé alakítani.
- A 2.2.5. fejezetben ismertetett lekérdező nyelv használatával megmérhetővé válik a forrásszöveg strukturális bonyolultsága, valamint a bonyolultság változása a programtranszformációs lépéseket követően.
- A bonyolult programrészek keresését végző algoritmus megmutatja azokat a problémákat, amelyek a strukturális bonyolultságot érintik, vagyis az előre definiált értékektől való eltérést számszerűen, minden transzformációs lépést követően.

Az elemző algoritmus e néhány tulajdonsága segítségünkre lehet egy olyan bonyolultság alapú transzformációs nyelv kidolgozásában, amellyel megoldhatóvá válnak a kézi erővel történő programtranszformációk okozta nehézségek.

A lekérdező nyelv nyelvtana tartalmazza a bonyolultság lekérdezésére alkalmas elemeket. Az automatizálás megvalósításához ki kell dolgoznunk azokat a nyelvi primitíveket, amelyekkel egyrészt a transzformációs lépéseket le lehet írni, másrészt a lekérdezések eredményeit lehet tárolni, valamint figyelni a transzformációs szkriptek futtatása közben.

Így megkapjuk a lekérdező nyelvtan egy kiterjesztett változatát, amellyel már lehetőségünk nyílik automatizált forrásszöveg javítási feladatok elvégzésére. (Példákat a 4.3 és a 4.2 példaprogramokban láthatunk.)

A nyelvi konstrukciók kidolgozását követően a lekérdező nyelv lexikális, szintaktikus és szemantikai elemzőjét fel kell készítenünk a kiterjesztett nyelvtan alapján történő elemzésére, majd meg kell konstruálnunk azt a fordítóprogramot, amely az algoritmus számára érthető nyelvre fordítja a nyelvtan alapján leírt transzformációs szkripteket.

Összefoglalva, a nyelv kiterjesztett változatára tekinthetünk úgy, mint transzformációs nyelvre, amely segítségével lehetőségünk nyílik olyan transzformációs lépések leírására, amelyek a forrásszöveget a bonyolultsági mértékek alapján átalakítják.

A transzformációs nyelvnek, és a hozzá tartozó elemző algoritmusnak a méréseket és a transzformációkat le kell fordítania a *RefactorErl* számára érthető nyelvre, és közben figyelnie kell a forrásszöveg állapotváltozásait, vagyis azt, hogy a szkriptekben leírtak alapján változik-e annak strukturális bonyolultsága.

Ehhez elsőként ellenőriznie kell a lexikális, valamint a szintaktikus struktúra helyességét, majd az ellenőrzést követően a bonyolultsági mértékek mérését és a transzformációs lépések futtatását kell megoldania.

Az elemzés legegyszerűbb módja az, ha az nyelvi elemek hatására az elemző a megfelelő paraméterekkel meghívja az adott transzformációs lépést megvalósító függvényt.

A bonyolultságot elemző algoritmus automatikusan méri, hogy a transzformációk megfelelő bonyolultságú forrásszöveget eredményeznek-e, így a transzformációs szkripteknek ezt már nem kell megtennie, csak el kell "olvasni" az elemző függvényétől kapott mérési eredményeket.

A nyelv a mérések, és a transzformációs lépések futtatása mellett lehetőséget biztosít olyan feltételek definiálására is (lásd: 4.3 és 4.2 programlisták) amelyek használatával a transzformációkat határokon belül tudjuk tartani. Ez a gyakorlatban azt jelenti, hogy meg tudjuk határozni, hogy egy lépés csak addig, vagy csak annyiszor fusson le, amíg egy meghatározott bonyolultsági értéket nem kapunk, vagy egy másik mérték rossz irányba nem változik.

A lefordított szkriptek futtatása szekvenciálisan történik, így a mérések és a transzformációk abban a sorrendben futnak le, ahogy a programban szerepelnek. Elsőként a méréseket tartalmazó szekció, majd ezt követi a transzformációs lépéseket leíró rész, ezután újra a mérések mindaddig, amíg a feltételek megengedik a további futást, vagy

el nem érjük a kívánt bonyolultsági értékeket (amely alapján átalakítjuk a forrásszöveget).

Természetesen a transzformációs nyelv használata és a futtatást végző algoritmus önmagukban nem biztosítják a megfelelő eredmény elérését, vagyis nem írják le a heurisztikát (az a programozó feladata), de használatukkal lehetőség nyílik annak leírására, így a kívánt eredmény elérését célzó program megalkotására.

A nyelv használatával megfogalmazhatjuk azokat a feltételeket, és definiálhatjuk a feltételek teljesülését célzó transzformációs lépéseket, amelyek az Erlang programokat automatikusan átalakítják a bonyolultsági mértékek változásai alapján.

Mikor optimalizálásról, vagy a forrásszöveg átstrukturálásáról beszélünk, nem minden esetben arra gondolunk, hogy hatékonyabb, vagy gyorsabban működő programot szeretnénk eredményül kapni.

Sokszor követni szeretnénk valamely számunkra előírt *layout*-ot, vagy a forrásszöveget akarjuk olvashatóbbá tenni, esetleg modulokra szeretnénk bontani azt, a benne szereplő függvény hívási gráfok alapján.

Előfordulhat az is, hogy egyszerűen csak követni akarjuk azokat az Erlang nyelvhez kidolgozott szabályokat, amelyeket a nyelv használói megalkottak, és jónak találtak. Mindez igaz akkor is, ha nem éppen egy fejlesztés közepén vagyunk, hanem egy korábban megírt program forrásszövegét szeretnénk a megfelelő formátumúra alakítani, vagy integrálni egy másik fejlesztéshez úgy, hogy betartjuk az ott meghatározott elvárásokat. Mindezekhez segítséget nyújthat a bonyolultság alapú transzformációs nyelv, de semmiképpen ne tekintsünk rá úgy, mint mesterséges intelligenciára, hanem csak mint egy eszközre, amelynek elemző és futtatást végző algoritmusa rendelkezik az alapvető logikával, ami lehetőséget biztosít az automatikus programkód átalakításhoz, valamint biztosítja az átalakítások elvégzéséhez szükséges méréseket és transzformációs lépéseket leíró programok készítését.

4.2. A fejezetben bemutatásra kerülő eredmények

A fejezetben bemutatjuk azt az általunk kifejlesztett, automatikus programtranszformációkra alkalmas nyelvet, amely segítségével a bonyolultsági mértékek javítására irányuló szkripteket készíthetünk. Ismertetjük a nyelv szintaxisát, és a futtatásához készített algoritmus működését, valamint megmutatjuk azokat a módszereket, amelyek segítségével az erlang programszövegek minőségét javíthatjuk a bonyolultsági mérőszámok alapján. A fejezetben bemutatott transzformációs nyelv működését ellenőriztük. Az ellenőrzés során kapott eredményeket a 6. fejezetben mutatjuk be.

4.2.1. Bonyolultság alapú transzformációs szkriptek

A transzformációs nyelv alkalmas arra, hogy a mérések alapján, előre definiált feltételek betartása mellett az $\mathcal{SG}$ szemantikus gráfban tárolt forráskódot automatikusan át lehessen alakítani (lásd: a 2. fejezetben), majd a gráfból az átalakítást követően a forrásszöveget újra elő tudjuk állítani.

A nyelvi elemek szintaxisába és az elemző algoritmusba be van építve a forrásszöveg transzformációját elindító és megállító feltételrendszer. A feltételek segítségével egyszerre több mérték mentén is átalakíthatjuk a programszöveget. Ez azt jelenti, hogy egy transzformációs lépést addig futtathatunk, amíg egy adott mérték a kellő értékűvé nem válik, de mellette figyelhetjük azt is, hogy a lépéssorozat ne változtathasson rossz irányba más mértékeket. Amennyiben ez mégis bekövetkezne, és a programban van a problémára nézve alternatív lépéssorozat, akkor az lefut.

A fejezetben bemutatunk néhány transzformációs szkriptet, amelyek az adott mérték alapján átalakítják a forrásszöveget. A példák és minta programok kidolgozása során nem törekszünk a teljességre, helyette inkább a nyelv lehetőségeinek az ismertetésére, mivel kifejlesztésének célja nem az összes lehetséges automatikus transzformációs lépés kidolgozása, hanem a lépések programjainak kidolgozására alkalmas algoritmus működésének ismertetése.

4.3. A transzformációs nyelv nyelvtana

A 2.2.5. fejezetben, a 2.11. ábrán a bonyolultsági mértékek mérésére készített lekérdező nyelv egyszerűsített, szkriptekben is alkalmazható változatát közöltük, a 4.1. ábrán pedig megmutatjuk annak a kiterjesztett változatát, amely optimalizáló szkriptek írását teszi lehetővé.

Az optimalizáló szkriptek használatával a programkódok minőségét automatikusan javítani tudjuk, a bonyolultsági mértékek változásait figyelembe véve.

A szintaxis leírásban a *Transformation* transzformációt (pl. `extract_fun`), a *Rel* relációt és egyéb operátort (pl. $<, <=, >=, >, like$), *LogCon* logikai operátort (pl. „és", „vagy"), a *CondValue* egész számot, vagy lexikális elemet jelölhet (pl. egy modul neve a *like* használata mellett). Az *Int* helyére a *limit* szakaszban pozitív egész szám helyettesíthető. (A nemterminális elemek nagy kezdőbetűsek, a nyelv kulcsszavai kis kezdőbetűvel lettek szedve.)

Az eredeti lekérdező nyelvtanhoz adott új vezérlő szerkezetek segítségével (*optimize, where, limit*), valamint a bonyolultsági mértékeket reprezentáló nevekkel, és a transzformációs lépések neveivel szkriptszerűen leírhatjuk az átalakításhoz használandó lépéssorozatot, illetve azokat a feltételeket, amelyeket alapul véve a futtató rendszer automatikusan elvégzi a keresési és a forráskód-transzformációs feladatokat.

Query → **MetricQuery** | **OptQuery**
OptQuery → **Opti Where Limit**
Opti → **optimize Transformation**
Transformation → *TransformationName*
| *TransformationName* **Params**
Params → (*Attr*, *ValueList*)
Where → **where Cond**
Cond → *Metric Rel CondValue*
| **Cond** *LogCon* **Cond**
Limit → **limit** *Int*

4.1. ábra. A transzformációs szkriptek teljes nyelvtana.

Az *optimize* szakaszban megadhatjuk az átalakításokhoz alkalmazandó transzformációs lépést és annak paramétereit.

A *where* kulcsszó után definiálható összetett feltétel az, amely a mérések elvégzését kezdeményezi, és a transzformációk lefutását vezérli, vagyis azt, hogy milyen feltételek teljesülése mellett kell az adott transzformációs lépést újra és újra lefuttatni, illetve azt is, hogy mely szemantikus gráf csomópontokat kell transzformálni. Az így megadható "bázisfeltételben", ami egy logikai kifejezés, szerepelnie kell legalább egy olyan részkifejezésnek, ami a transzformációra kijelölt modulokon, vagy függvényeken mérhető bonyolultsági mértéket, egy aritmetikai operátort, valamint egy konstans értéket tartalmaz.

A kijelölt, vagyis a transzformáció tárgyát képező elemek így már nem direkt definiált programkonstrukciók, vagy kifejezések, hanem a feltételek alapján automatikusan kiválasztásra kerülő szemantikus gráf csomópontok. Ezzel a módszerrel a transzformálandó programrészek kijelölése a lexikális szintről átkerül a szemantikus elemzések szintjére.

A transzformációs nyelven leírt szkript futása során az elemző megkeresi a feltételének megfelelő programrészeket, majd végrehajtja a *optimize* szakaszban kijelölt transzformációt, ezután megméri a feltételben adott bonyolultsági mértékeket az összes szemantikus gráf csomópontra. Amennyiben egy csomópontot az operátor és a konstans alapján már nem kell transzformálni, akkor az kiesik a szkript hatásköréből. Amennyiben nincs olyan csomópont, amelyen a transzformációt újra le kell futtatni, a szkript futása megáll.

A transzformációk alkalmazásával nem minden esetben jutunk el a kitűzött célhoz, vagyis a szkriptet újra és újra futtatva, az mindig talál újabb transzformációra váró gráf csomópontokat (néha a szkript hozza létre azokat a transzformációk alkalmazásával).

Ezen feltételek mellett előfordulhatnának olyan esetek, mikor a szkript futása nem áll meg. A probléma elkerülésére a *limit* kulcsszó után leírt konstans segítségével definiálható a maximálisan futtatható ismétlések száma. Amennyiben tehát a transzfor-

mációs lépés nem hozza meg a kívánt eredményt, a *limit* konstansa adott lépésszám után biztosan megállítja a futását.

Ezen a ponton már nem a nyelvtani elemek megalkotása, a transzformációk futtatása, vagy a feltételek kiértékeléséhez a bonyolultsági mértékek mérése jelenti a problémát, hanem az adott transzformációs lépés futásához szükséges paraméterek összeállítása. A transzformációk paraméterei a legtöbb esetben a következők:

- A modul, vagy modulok, amelyek tartalmazzák a transzformációban szereplő programelemeket. Modulra és függvény típusra alkalmazott transzformációk esetén is.

- A transzformációra kijelölt csomópontok lexikális szinten adott pozíciója, amely lehet sor-oszlop pár, vagy karakterpozíció. A bonyolultsági mértékek modulokra és függvényekre vannak definiálva, így a mérések során az elemző csak azok szemantikus gráf csomópontjait tárolja. Tehát, ha egy adott mértéktől függő feltételben megmérjük a bonyolultságot, akkor a szkript a feltételnek megfelelő függvényt, vagy a modult reprezentáló csúcsot ad eredményül. A csomópontból kiindulva kell megtalálni annak pozícióját forrásszövegben ahhoz, hogy transzformálhassuk.

- Ezen kívül szükség van még az adott transzformációra jellemző egyéb paraméterekre, mint a függvény neve, paraméterszáma, stb.

A probléma megértéséhez vegyük azt az esetet, amikor egy összetett kifejezésben szereplő, legmélyebben beágyazott részkifejezést kell kiemelnünk az *extract_function* (vagy másik nevén *intruduce_function*) transzformáció alkalmazásával.

Tegyük fel, hogy a feltétel, ami alapján megtaláltuk az adott függvény csomópontot az, hogy a legnagyobb megengedett beágyazottsági szintje a *case* kifejezéseknek három, és a megengedett maximális lépésszám szintén három (az esetet leíró szkriptet a 4.3 forrásszövegben láthatjuk). Ekkor a szkript lexikális, szintaktikai és szemantikai elemzését követően meg kell mérni az összes, az $\mathcal{SG}$ szemantikus gráfban található modul függvényein a *max_depth_of_cases* bonyolultsági mértéket, majd a feltételben szereplő értéknél nagyobb beágyazottságot mutató függvényeket ki kell gyűjteni, egy listába az őket jellemző beágyazottsági mértékekkel együtt:

$$(function_1, 5), (function_2, 6), ...(function_n, 4).$$

Ezt követően meg kell keresni a legbelső *case* kifejezést minden függvény csomópontban (paraméterezhető, hogy az adott kiemelésnél a legbelső, vagy a legkülső kifejezést emeljük ki.) (4.2 forrásszöveg). Ezek a kifejezések valójában szintén mind szemantikus gráf csomópontok, és rendelkeznek olyan lexikális attribútumokkal, mint a fájlbeli pozíció.

Alkalmazni kell minden elemre a szkriptben leírt transzformációt, amely mindegyik kiemelendő *case* kifejezéshez egy új függvényt hoz létre (ezek nevének előtagját paraméterként megadhatja a szkript írója (lásd később), vagy az algoritmus véletlenszerűen generálja azokat).

A futtatás során minden futási ciklusban ellenőrizni kell azt is, hogy a megadott maximális lépésszámot túlléptük-e már. Ha nem, akkor a lépéssorozat futhat tovább, ellenkező esetben meg kell állnia.

```
f1() ->
 case ...
    case ...
       case ...
           [case ... end]
       end
    end
 end.
```

4.2. ábra. Legbelső *case* kifejezés

A minden "következő" iteráló lépésben a feltétel alapján a méréseket újra el kell végezni minden lehetséges csomópontra (az előzőekben kiesetteken is, mert a transzformációk alakíthatják úgy a forráskódot, hogy egyes csomópontok az összetett feltételek alapján újra a szkript hatókörébe kerülhetnek).

```
optimize
  extract_function
where
  max_depth_of_cases > 3
limit 3
```

4.3. ábra. Példa szkript

Ezután jöhet a következő kör, mindaddig, amíg a feltételek alapján már nem található transzformálásra váró csomópont, vagy a maximális lépésszámban (*limit*) definiált konstans értékét el nem érjük.

A 4.3. példában szereplő transzformációban a feltételnek nem megfelelő függvények legbelső *case* kifejezését kell megtalálni. Két külön álló *case* kifejezés esetén, ahol az egyik egy, a másik két mélységben van beágyazva, a szkript a mérésekhez hasonlóan a legmélyebben beágyazottat találja meg, így azt is fogja kiemelni.

Más transzformációk és mértékek szkriptekben való alkalmazása esetén egészen más paraméterekre van szükség, és minden esetben többre is.

```
-module(deep).
-export([f/1]).

f(A) ->
  case A of
      1 -> 1;
      2 -> 2
  end,
  case A of
      3 -> case A of
                  4 -> 4;
              5 -> 5
           end;
      6 -> 6
 end.
```

4.4. ábra. Több *case* egy függvényben

A *rename_function* transzformációnak a függvények nevét és a paramétereiknek a számát kell megkeresnie, ami a függvény csomópontból egyértelműen azonosítható, az *introduce_record*-nak a függvény formális paraméterlistájára van szüksége, és így tovább.

Az algoritmusnak ezért minden transzformáció típushoz össze kell gyűjtenie az arra jellemző paramétereket az adott függvény, vagy a modul csomópontból kiindulva, vagyis mindegyik transzformációs lépéshez meg kell konstruálni azokat a szemantikus gráfon definiált útvonal kifejezéseket (az útvonal kifejezések és azok futtatásához szükséges algoritmust korábban már ismertettük, (lásd: a 2. fejezetben), amelyek a paraméterlistához szükséges adatokat szolgáltatják.

4.3.1. Az *optimize* szakasz

A szkriptek három fő szakasza közül az első az *optimize*. Ebben a szakaszban adjuk meg az alkalmazni kívánt transzformációs lépést - egy *optimize* pontosan egy transzformációt tartalmazhat - és annak paramétereit, amennyiben erre szükség van.

A 4.5 példában a *rename_function* lépést paramétereztük egy tetszőleges modul nevével. A szkript *where* szakaszában azokat a függvényeket szűrtük ki, amelyek neveinek a karakterekben mért hossza nagyobb, mint harminc karakter.

Az új függvények neve a *newname* atommal fog kezdődni ha több függvény is fennakad a szűrőn, a neveik meg lesznek sorszámozva. Amennyiben az átnevezés tárgyát

képző modulban már van a generálttal megegyező nevű függvény, a szkript azt nem nevezi át.

Paraméteres transzformáció

```
optimize
  rename_function [mod, newname]
where
  length_of_name > 30
limit 1
```

4.5. ábra. Paraméteres transzformáció

Az ilyen jellegű átalakításokat akkor használhatjuk sikerrel, ha egy korábbi transzformációs lépéssorozat új függvényeket hozott be a modulokba, amelyek generált neve hosszú, vagy számunkra nem megfelelő.

4.3.2. Paraméterek a transzformációkhoz

Az *optimize* szakaszban definiált transzformációs lépések paraméterei tehát függenek a transzformáció típusától. Ezért, ahogy azt korábban, a bonyolult programrészek lokalizálásáról szóló részben (a 3. fejezet) már megtettük, vegyük sorra a fontosabb transzformációs lépéseket, valamint azok paramétereit. Az elemzést most úgy kell elvégeznünk, hogy a szkriptek paraméterezési lehetőségeit tartjuk szem előtt.

4.3.3. A transzformációk általános paraméterei

A függvény átnevezése (*rename_function*) lépés paraméterként várja az átnevezésre váró függvény modulját, és magát a függvényt, ami lehet egy szemantikus gráfbeli függvény csomópont.

A csomóponthoz néhány egyszerű lépéssel megtalálhatjuk a függvényhez tartozó nevet, valamint a paraméterszámot. Szükség van még a függvény új nevére, ami jelen esetben lehet generált név, vagy egy függvény esetén megadott (a szkript megfelelő részén definiált) név.

A felsorolt paraméterek közül a függvény csomópont rendelkezésre áll, mivel ez a bonyolultsági mértékek mérésekor a feltételek alapján kigyűjtésre került. A függvényből kiindulva annak modulja egyszerűen megtalálható a $[\{func, back\}]$ útvonal kifejezés alkalmazásával (gráf útvonalakról bővebben a 2. fejezetben olvashatunk).

A (*reorder_function_parameters*) függvény paraméterek sorrendjének felcseréléséhez meg kell adni a paraméter cserére kijelölt függvényt tartalmazó modult és a függvényt. Ezután a két csomópont alapján meg kell keresni a függvény formális paraméterlistáját és megvizsgálni az elemek pozícióját (ez a keresés nyilvánvalóan szintén a

szemantikus gráfban leírt információk alapján, útvonal kifejezések alkalmazásával történik). A pozíció itt nem a paramétereknek a forrásszövegben elfoglalt helyét jelenti, hanem a paraméterlista elemeinek egymáshoz viszonyított sorrendjét.

Végül a transzformáció elvégzéséhez szükség van az új sorrendre, hogy a cseréket végre lehessen hajtani.

Minden transzformációs lépésnél, amely függvényeket módosít, rendelkezésre áll a függvény csomópontja, abból pedig minden, a transzformációs lépéshez szükséges tulajdonság, vagy más kapcsolódó csomópont elérhető. A modulokra vonatkozó transzformációknál is ugyan ez a helyzet, csak a kiindulási elem (az $\mathcal{SG}$ gráfban) egy modul csomópontja. A modulok, vagy a fejléc fájlok átnevezéséhez az azokat reprezentáló gráfcsomópont mellet szükség van az új névre (ez szintén generálható).

A függvény mozgatása (*move_function*) transzformáció a paraméterezést tekintve egyszerű, mivel a mozgatott függvény csomópontok mellett csak a cél modul azonosítására van szükség (ha ez nem létezik, akkor generálunk újat és elhelyezzük azt a szemantikus gráfban).

A rekord, vagy a makró mozgatásánál is (*move_record*, *move_macro*) hasonló a helyzet, mint a függvényeknél, csak itt a modul csomópontokból indulunk ki, és azok makróit, vagy rekordjait vesszük sorra egymás után, és megfelelő számban átmozgatjuk őket a cél modulba (vagy fejléc fájlba). A cél fájlok ebben az esetben is generálhatóak.

A kifejezés kiemelése (*extract_function*) a paraméterek gyűjtése terén érdekes lépés, mivel itt a függvényben található beágyazott kifejezést kell megtalálni, és kiemelni azt egy új függvénybe (az is egy paraméter, hogy a legmélyebben találhatót, vagy a külsőt keressük). A kiemelés során az új függvények nevére van szükség, és számos esetben automatikusan generált formális paraméterlistára, amely gondoskodik a megfelelő paraméterátadásról a régi és a létrejövő új függvény között. A függvény neveket lehet generálni, vagy egy függvény esetén megadni (ez nagyon ritka eset egy szkriptben, és kizárólag a feltételeket nem tartalmazó transzformációk esetén fordulhat elő).

A beágyazott kifejezés megtalálása hasonló a feltételeket leíró részben, vagyis a bonyolultságmérés során alkalmazott függvényben (*max_depth_of_cases*) található lépésekkel. Ebből az okból kifolyólag érdemes már a méréskor elhelyezni az aktuálisan megtalált és legmélyebben fekvő kifejezés gráf csomópontját egy erre a célra kialakított tárban, majd a transzformációk alkalmazása során felhasználni azt.

A rekord bevezetése (*introduce_record*) lépés alkalmazásához adott a transzformálandó függvény, illetve annak paraméterlistája. Alapértelmezés szerint az összes paraméternek szerepelnie kell az új rekordban, de megadható az is, hogy mely (összefüggő sorozat) paramétercsoportot tartalmazza a létrejövő új struktúra. A transzformációt szkriptekben csak a függvény paraméterlistájára lehet alkalmazni, a függvény törzsében elhelyezett rekordra való hivatkozások (*application*) esetén nem.

A paraméterek rendezett n-esre alakítása, vagyis a *introduce_tuple* nevű lépés paraméterezéséhez szükségünk van a függvényre, amely paramétereit transzformáljuk (ez itt is adott), és meg kell keresni a paraméterlistájában szereplő elemeket reprezentáló gráf csomópontokat a megfelelő szemantikus gráf éleken keresztül. Ha ez megvan, létre kell hozni az új adatstruktúrát (*tuple*) és hozzá kell kötni a megfelelő címkéjű élekkel a paraméterlista elemeit.

A kifejezés kötése változóba (*introduce_variable*), a lépés paraméterei, a függvény csomópont és ebből kiindulva az első olyan kifejezés, amely többször előfordul. A függvény csomópontból elindulva meg kell keresni az összes, egyszerű kifejezést reprezentáló szemantikus gráf csomópontot (az egyszerű ebben az esetben azt jelenti, hogy nem vezérlő szerkezetről, vagy függvény kifejezésről van szó), majd megnézni, hogy melyik fordul elő a legtöbbször. Ha megtaláltuk, akkor kötni kell a kifejezést egy generált nevű változóba, majd lecserélni a kifejezés előfordulásait erre az új elemere. Nyilván a cserét csak akkor lehet végrehajtani, ha ezt a transzformáció szabályai nem tiltják. Minden transzformációs lépés tartalmazza a rá vonatkozó szabályokat, és csak akkor hajtódik végre, ha az adott lépés nem sérti azokat.

A függvényhívás eliminálása (*eliminate_function_call*) transzformáció paraméterei a transzformálandó függvény, és az abban szereplő első függvényhívás, amit lecserélünk a hívott függvényben szereplő függvénytörzsre.

A függvények hívását elimináló lépéshez hasonlóan minden eliminálást végző transzformációhoz a cserélendő kifejezést kell detektálni. Kivételt képez ez alól a változó eliminálása (*eliminate_variable*), ahol az első kifejezést kötő változóhoz tartozó csomópontot kell megtalálni és a benne kötött kifejezésre lecserélni.

4.3.4. A transzformációs lépésekben alkalmazható paraméterek

Az *optimize* szakaszban, ahogy azt korábban már láthattuk (4. fejezet), számos, a *RefactorErl*-ben elérhető transzformációs lépést meg tudunk hívni a forrásszöveg átalakításához.

A különböző transzformációk felhasználói interfészen elérhető változataihoz képest a szkriptekben leírtak többféle paraméterrel rendelkeznek, illetve tartalmaznak olyan paraméterezési lehetőségeket, amelyek bevezetését a felhasználói interakció hiánya indokolta. A különböző lépések elvégzése előtt és közben nem lehet megszerezni semmilyen információt a felmerülő döntéshelyzetek megoldására.

Vegyük azt az egyszerű példát, mikor egy függvény átnevezésekor olyan függvénynevet adunk meg, amely már létezik a transzformálandó függvény moduljában (és az ilyen nevű függvény paramétereinek száma megegyezik az átnevezendő függvényével). A legtöbb transzformációs lépésnél felmerülhetnek ilyen és ehhez hasonló problémák, amelyeket meg kell oldanunk ahhoz, hogy a szkripteket futtatni lehessen.

Átnevezések. A *rename_fun* és a *rename_variable*, valamint az összes átnevezést végző transzformáció számára megadhatjuk az új nevet *(name, "nev")*, a név prefixét a *(prefix, 'prefix')* párral. A függvényekre, a párban az *f* előtagot használhatjuk *(fprefix, 'fprefix')*. Így egyrészt az átnevezések során megadhatunk a generált nevekhez módosítókat, vagy definiálhatjuk a teljes nevet. Nyilván változók esetén az előtagot, vagy a teljes nevet sztring, a függvények esetén atom típusban kell megadnunk, ahogy ezt a 4.6 példaprogramban láthatjuk.

```
optimize
  rename_mod (prefix, "lib_")
where
  ...

optimize
  rename_fun (fprefix, 'lib_fun_')
where
  ...

optimize
  rename_fun (name, 'lib_fun_')
where
```

4.6. ábra. Átnevezés paraméterei

Kifejezés kiemelése. A *extract_function*, vagyis a kifejezés kiemelése transzformáció lehetséges paraméterei a kiemelésre szánt kifejezések típusa (lásd: a 4.7. szkriptben). Mikor nem adunk meg típust, akkor az alapértelmezett *all* paraméter érvényesül. Ekkor minden beágyazott kifejezést sorra kell venni és ezek közül kiemelni a legbelsőt, vagy a külsőt. A kiemelés szintje szintén paraméterként adható meg (*pos, last/first*).

A transzformáció ezek alapján sorra veszi az adott függvény ágban található egymásba ágyazott kifejezéseket. Amennyiben egymást követően több, azonos beágyazottsággal rendelkező konstrukciót talál, az elsőt veszi, majd bejárja azt (megkeresi a legbelső kifejezést), és kiemeli úgy, hogy létrehoz egy új függvényt, majd generál számára egy nevet. A függvényhívást elhelyezi a kifejezés korábbi helyén (lásd: az 3. fejezetben). A fentiek alapján a paraméter a követező kifejezéstípusokat tartalmazhatja:

[*(expr_type, all) (expr_type, case_expr) (expr_type, if_expr) (expr_type, fun_expr) (expr_type, try_expr)*]

```
optimize
  extract_fun (expr_type, case_expr)
where
  max_depth_of_cases > 3
  ...

optimize
  extract_fun
where
  max_depth_of_cases > 3
  ...
```

4.7. ábra. Kifejezés kiemelése paraméterei

Ezek a paraméterek meghatározzák, hogy mely kifejezés típus beágyazottságát számoljuk és emeljük ki. Amennyiben egy szkriptben a *case* kifejezések beágyazottságát vizsgáljuk, de a kiemeléshez *if* kifejezést adunk meg, a kiemelés nem történik meg. Ha viszont az *all* paraméter aktív, bármely kifejezés típust kiemeli a szkript.

Függvények és más programelemek mozgatása. A mozgatásokat végző transzformációkhoz megadhatunk cél modult, ahová a szkript által kiválasztott programelemeket mozgatni lehet. Alapértelmezésként - mikor nem adunk meg cél modult -, a szkript létrehoz egy *target_mod* nevű modult a munkakönyvtárba (ahol éppen dolgozik), majd hozzáadja a szemantikus gráfhoz ($\mathcal{SG}$ lásd: a 2 fejezetben), majd ezt a modult használja az átmozgatások céljaként.

Ha nem akarjuk az alapértelmezett modult használni, megadhatunk létező, vagy létrehozandó cél modult. Amennyiben a cél nem létezik, az imént bemutatott módon a szkript létrehozza.

A paramétert a következő formában kell megadnunk: (*targetmod, 'terget_mod'*), ahol a rendezett n-es második eleme a cél modul neve. A név kiegészül a *.erl* kiterjesztéssel.

Függvény paraméterek átrendezése. A *reorder_funpar* transzformációnak, ha szkriptekben alkalmazzuk, akkor szüksége van a paraméterek új sorrendjére, amely alapján átrendezheti azokat. A sorrend egy rendezett n-essel adható meg, mint a *(order, [x, y])*, ahol a pár első eleme a transzformáció neve, a második pedig a sorrend. A 4.9. példában a *[2, 1]* listával arra utasítjuk a programot, hogy az első paramétert cserélje fel a másodikkal, vagyis az első paraméter kerüljön a második helyre, a második az első helyre. Amennyiben nem adunk meg semmilyen paramétert, a paraméterek

```
optimize
  move_fun (targetmod, 'target')
where
  ...

optimize
  move_macro (targetmod, 'header')
where
  ...

optimize
  move_rec (targetmod, 'header')
where
  number_of_records > 0
  ...
```

4.8. ábra. Programelemek mozgatásának paraméterei

```
optimize
  move_fun
where
  number_of_funpar > 2
  ...

optimize
  reorder_funpar (order, [2, 1])
where
  number_of_funpar == 2
  ...
```

4.9. ábra. Függvény paraméterek sorrendjének a cseréje

számától függetlenül, minden elem a helyén marad a transzformáció során. Az elemző ilyenkor megszámolja a paramétereket, és a sorszámukat helyezi el a listában, ami helybenhagyást eredményez.

4.3.5. A *where* szakasz

A szkriptek *where* szakaszában megadott feltételek szűrik a szemantikus gráfban tárolt programszöveget reprezentáló csúcsokat, vagyis kiválogatják azokat a modul és függ-

vény csomópontokat, amelyek a feltételek alapján a transzformációban részt kell, hogy vegyenek.

Modulokra és függvényekre vonatkozó feltételek

```
optimize
  move_function [targetmod]
where
  number_of_function > 30
   and
  number_of_funclauses > 5
limit 8
```

4.10. ábra. Függvények mozgatása feltételek alapján

A szoftverbonyolultsági mértékek némelyike kizárólag függvények, egyesek modulok, és más mértékek mindkét típus tulajdonságait mérik. A *where* szakaszban szereplő feltételek egyaránt vonatkozhatnak modulokra, és függvényekre is. Pontosabban a bonyolultsági mértékek alapján az alábbi lehetőségek adódnak:

- Amennyiben a feltételben kizárólag modulra vonatkozó bonyolultsági mértékek szerepelnek, modul típusú csomópontok kerülnek a szkript hatókörébe. Ilyen esetekben a megelőző *optimize* szakaszban olyan transzformációkat érdemes szerepeltetni, amelyek modulokat módosítanak, mivel a függvényekre vonatkozók nem futnak le, és ennek következtében a szkriptben maradás feltételei mindig teljesülnek, így annak futását csak a *limit* szakaszban definiált konstans értéke állítja meg.

  A 4.11 szkript *optimize* részében a *rename_module* transzformációt kell alkalmazni minden olyan modulra, ahol a függvények száma nagyobb, mint tíz (*number_of_functions* > 9), és nincs a modulból kifelé irányuló függvényhívás (*function_calls_out* < 1). A feltételek azt is leírják, hogy találnunk kell kívülről érkező hívást (*function_calls_in* > 4) legalább a függvények felére. A transzformáció paramétere alapján minden, a feltételrendszernek megfelelő modul neve az átnevezés során megkapja a *lib_* előtagot, mivel ezeket a modulokat a példában *library* moduloknak tekintjük.

- Ha a feltételes részben modulra, és függvényre vonatkozó mértékek vannak definiálva - nem olyanok, amelyek mindkét típusra vonatkoznak -, elsőként a feltételnek megfelelő modulok kerülnek be a hatókörbe, majd az ilyen módon gyűjtött modulok függvényei (más modulok függvényei nem).

Amennyiben egy modul kikerül a gyűjteményből, annak függvényei sem kerülnek be újra, hiába felelnek meg a függvényekre vonatkozó feltételek valamelyikének, viszont, ha a modul újra a látókörbe kerül, a függvényit ismét magával hozza.

A 4.10 példa szkriptben a *move_function* transzformációt hajtjuk végre azokra a függvényekre, amelyek harmincnál több függvényt tartalmazó modulokban találhatóak és a *clause*-aik száma több mint öt.

A feltételen fennakadó függvényeket a *targetmod* nevű modulba mozgatjuk. Amennyiben a modul nem létezik, a szkript létrehozza. Ha nem adunk meg cél modult, akkor létrejön egy un. *default* modul ugyanilyen névvel, és a függvények ebbe a modulba kerülnek át. (A következő, más szkriptben definiált, de célmodult nem tartalmazó mozgatás során ebbe a *default* modulba helyezi el a függvényeket.)

A *limit* konstans értéke miatt a szkriptet maximum nyolc lépésben futtathatjuk (ha egyébként a feltételek miatt nem állna meg).

- Abban az esetben, ha modulra és függvényekre egyaránt alkalmazható bonyolultsági mértékeket definiálunk a *where* szakaszban, az összes modul azon függvényei kerülnek a gyűjteménybe, amelyeket a függvényekre alkalmazott feltételek kiszűrnek. Ez azért van így, mert a mindkét típusra vonatkozó mértékek általában függvény típust mérnek, és modulra alkalmazva azokat az összes, a modulban szereplő függvényen mért értékeket összesítik.

- Kizárólag függvényekre vonatkozó mértékek alkalmazása mellett a szemantikus gráfban található összes modul függvényeit szűrjük. Ha egy kijelölt modul függvényeit szeretnénk mérni, akkor a feltételek között meg kell adnunk egyet, ami a modul nevét vizsgálja (*name like 'nev_'*, vagy *name == 'nev'*).

  A 4.12 példa szkriptben minden olyan függvényt átmozgatunk egy *library_mod* nevű modulba, amelyek nem hívnak a modulon kívülről függvényeket (*internal_call_from < 1*), de hívják őket más modulok függvényei *(external_calls > 0)*. (A mozgatásoknál a cél modult nem vizsgálja a feltétel.)

- Definiálhatunk feltétel nélküli transzformációs lépéseket is. Ennek az a módja, hogy a *where* szakaszt teljesen elhagyjuk (lásd: 4.13 forrásszöveg). A feltétel nélküli transzformációs lépések használatával nagy mennyiségű forrásszövegben javíthatunk tipikus programozói hibákat, vagy lecserélhetünk *deprikálásra* váró, esetleg a nyelv adott verziójában nem létező program konstrukciókat.

A modul és függvény csomópontok gyűjtését lehet specializálni a *where* szakaszban megfogalmazott feltételben szereplő bonyolultsági mértékek elejének a kiegészítésével. Ez azt jelenti, hogy a mérték nevét elláthatjuk az *m_*, vagy az *f_* előtaggal. A mo-

dulra és függvényre is egyaránt alkalmazható mértékek esetén *m_* előtag csak modulok mérését kezdeményezi, az *f_* csak függvényekét.

Modulokra vonatkozó feltételek

```
optimize
  rename_module [prefix:lib_]
where
   function_calls_out < 1
  and
   number_of_functions > 9
  and
   function_calls_in > 5
limit 8
```

4.11. ábra. Függvények mozgatása feltételek alapján

A szkriptek elemeit és az algoritmust, ami vezérli szükség esetén (ha nem tudunk egy összetettebb feladatot megoldani a mértékek és a transzformációk segítségével), a fenti lehetőségek mellett kiegészíthetjük újabbakkal. A kiegészítés három ponton történhet. Az első, hogy új transzformációkat implementálunk. Ekkor az *optimize* szakaszt lehet bővíteni, valamint - ahogy azt a *rename_function* transzformációnál láthattuk (lásd: 4.5 példa) -, az eredeti funkciót ki tudjuk egészíteni paraméterekkel. (A 4.11 szkriptben is arra használtuk az átnevezést, hogy a modulok nevéhez előtagot illesszünk.)

Függvényekre vonatkozó feltételek

```
optimize
  move_function [library_mod]
where
   external_calls > 0
  and
   internal_calls < 1
limit 8
```

4.12. ábra. *Library* modulok függvényeinek gyűjtése

A második lehetőség az, hogy új bonyolultsági mértékeket vezetünk be, így bővítve a *where* szakasz funkcionalitását. A 4.12. szkriptben láthatjuk, hogy a már létező mértékeket tudjuk specializálni, és ezekből új mértékeket készíteni *(*internal_calls, *external_calls*).

A harmadik, de lényegesen bonyolultabb bővítés során a szkripteket újabb szakaszokkal egészíthetjük ki, de ebben az esetben a nyelv lexikális és szintaktikai elemzőjét át kell alakítanunk, míg az előző átalakítások mellett erre a lépésre nincs szükség.

Feltétel nélküli szkript

```
optimize rename_module [prefix:work_]
```

4.13. ábra. Modulok átnevezése

4.4. Az algoritmus működésének elemzése

Az 4.15 forrásszöveget használjuk példaprogramként. Ezt a programszöveget fogjuk transzformálni a 4.3 szkripthez hasonló, de annál több feltételt tartalmazó transzformációs lépéssorozat használatával (a szkriptet a 4.14 listában találjuk).

A forrásszövegben szereplő modulnak egy függvénye *f/1*, ami háromszoros mélységben egymásba ágyazott *case* kifejezéseket tartalmaz.

Mielőtt elemeznénk a szkript működését, vizsgáljuk meg a sorok jelentését, vagyis azt, hogy az egyes programelemek külön-külön mit jelentenek. Az első részben, az *optimize* kulcsszó után megadtuk a transzformáció nevét - *extract_function* paraméterek nélkül, általános formában - ,amelyet futtatni kell a feltétel alapján megjelenő szemantikus gráf csomópontokra (programrészekre).

A második szakasz, a *where* kulcsszót követően tartalmazza azokat a feltételeket, amelyek a szkript futását engedik, vagy éppen tiltják.

Az első feltétel a *max_depth_of_cases* > *1*, a második a *number_of_functions* < *10*, ami azt jelenti, hogy a transzformációt mindaddig ismételni kell az adott gráf csomópontokon, amíg a függvények *case* kifejezéseire jellemző legnagyobb beágyazottsági szint egy, és a modulban szereplő függvények száma kisebb mint tíz.

A két feltétel együttes alkalmazása azért érdekes, mert az *extract_function* transzformáció a kijelölt kifejezésekből új függvényeket állít elő, így növeli a függvényszámot a modulban.

```
optimiz
  extract_function
where
  max_depth_of_cases > 1
 and
  number_of_function < 10
limit 2
```

4.14. ábra. Beágyazott kifejezés kiemelése

A szkript harmadik szakaszában a *limit* értéke 2. Ez az érték adja meg a szkript futásának, vagyis a transzformációk alkalmazásának lehetséges maximum számát. A limit megadása nem kötelező, de ekkor az alapértelmezett érték lép életbe.

A három szakasz két lépéssorozatot ír le: az első lépés kiszámítja a legnagyobb szintű *case* egymásba ágyazást a vizsgált modulban.

A második lépés elindítja a transzformációt végző algoritmust, ami kísérletet tesz azon szemantikus gráfbeli csomópontok számának a csökkentésére, amelyekre a feltétel teljesül.

A *number_of_functions* csak egy része a feltételnek, a szkript kiválasztja azokat a gráf csomópontokat (függvény csomópontokról van szó), ahol *max_depth_of_cases* nagyobb, mint egy.

A kiindulási forráskódban az *f/1* függvény tartalmaz egy hárommélységű *case* vezérlő szerkezetet, ezen a ponton tehát refaktorálni kell a forrásszöveget egy új függvény bevezetésével (*extract_function*). A transzformáció veszi a legbelsőbb *case* szerkezetet leíró kifejezés csomópontját, majd kiemeli azt egy új, *f0/1* (generált) nevű függvénybe.

A feltétel újra kiértékelődik (újra megmérjük a lehetséges csomópontokat az adott mérték alapján): azt kapjuk, hogy a *number_of_functions*, vagyis a függvények száma a modulban kettőre nőtt, és a beágyazottság maximuma, a *max_depth_of_cases* kettőre csökkent. Mivel még nem értük el a lépéskorlátot (*limit*), vagyis a maximális lépések számát, és a feltételek is lehetővé teszik a futást, a szkript tovább dolgozik. Újra lefut a transzformációk sorozata (ahány transzformálandó csomópont, annyi lépés, de jelenleg egy): ekkor elérjük a transzformáló lépések maximális felső határát és ezzel együtt megszűnik az összes olyan pont, ahol a feltétel teljesült, tehát a szkript megáll. Ebben az esetben a lépésszám (*limit*) állítja meg, mivel ez minden esetben a legerősebb feltétel.

Mivel a transzformációt vezérlő algoritmus végrehajtja a szkriptet külső beavatkozás nélkül, előfordulhat, hogy a végeredményt tekintve a forráskód nem lesz jobb az eredetinél. Ha a transzformációk végrehajtását követően az eredményül kapott program elvárason aluli, lehetőségünk van visszatérni a szkript lefuttatása előtti állapothoz.

Számos esetben születhet rossz eredmény a lépések futtatása során. Mindez azért van így, mert az adott szkript automatikusan végrehajtja a benne leírtakat, és nem rendelkezik komolyabb intelligenciával.

A tudást a szkript írója birtokolja, és az általa leírtak történnek meg a szkript futtatása során. Nincs ez máshogy itt sem, mint az általános célú programozási nyelvek esetében. Amennyiben rossz programot írunk, az rossz eredményt fog szolgáltatni, és előfordulhat az is, hogy a program jó, de az adott inputra nem képes elfogadható eredményt produkálni. Ilyenkor vagy a szkriptet kell átalakítani, vagy a bemenetét megváltoztatni.

Az eredeti forrásszöveg

```
f({A, B})->
  case A of
    send -> case B of
      {Pid, Data} ->
         case Pid of
           {pid, P} -> P ! Data;
           _        -> Pid ! B end;
       _   -> null ! B end};
    _ -> mod:wait() end.
```

↓

Az első lépés eredménye

```
f({A, B})->
  case A of
      send -> f0(B);
    _ -> mod:wait() end.

f0(B) ->
  case B of
   {Pid, Data} ->
      case Pid of
         {pid, P} -> P ! Data;
         _        -> Pid ! B end;
  _ -> null ! B end.
```

↓

A teljes szkript futásának az eredménye

```
f({A, B})->
  case A of
        send -> f0(B);
           _ -> mod:wait() end.
f0(B) ->
    case B of
        {Pid, Data} -> f1(B, Data, Pid);
                 _ -> {null ! B} end.
f1(B, Data, Pid) ->
  case Pid of
      {pid, P} -> P ! Data;
             _ -> Pid ! B end.
```

4.15. ábra. Automatikus transzformációs lépések

Amennyiben az eredmény megfelelő, de további transzformációkat szeretnénk végrehajtani, a szkriptet kiegészíthetjük újabb szakaszokkal, amelyek szintén tartalmazzák az *optimize*, a *where* és a *limit* részeket.

A futtatás szekvenciálisan történik a szakaszokra nézve is. Miután az első *optimize* szakasz lefutott és mindent átalakított, amire jogosítványa volt, jön a következő, amely szintén végrehajtja a feladatait.

4.5. Transzformációs szkriptek alkalmazása

Ahogy láthattuk, a forrásszöveg átalakítására kifejlesztett algoritmus és transzformációs nyelv felhasználásával bármely, általunk definiált szoftverbonyolultsági mérték (több mértékre is) mentén optimalizálhatjuk az Erlang programok forrásszövegét.

Hogy ezt az állítást igazoljuk, megkísérlünk egy, az ismert *McCabe* bonyolultsági mérték, vagyis a McCabe-féle ciklomatikus szám (a definíció a 2.1.7. fejezetben) javítását célzó szkriptet készíteni a fejezet első szakaszában ismertetett nyelven, és lefuttatni azt az Erlang disztribúciókba integrált, ismert szoftver komponensekre.

A teszteléshez azért a *McCabe*- féle ciklomatikus számot választottuk, mert ez a mérték elég ismert ahhoz, hogy ne csak Erlang, vagy egyéb funkcionális nyelveket ismerő programozók számára nyújtson elegendő információt a programszöveg bonyolultságának mértékéről.

A mérőszám megválasztása mellett, a forrásszöveg kiválasztásánál két szempontot vettünk figyelembe. Az egyik a hozzáférhetőség, a másik a méret. Az első szempont azért fontos, mert olyan eredményeket akarunk előállítani, amelyeket bárki reprodukálhat. A másik szempont a szoftver mérete, a reprezentatív eredmények elérése érdekében.

Az általunk mért szoftver a *Dialyzer*, az Erlang része, amely szoftver elég bonyolult ahhoz, hogy a vizsgált mértékek mindegyikére adjon elemezhető eredményeket. Összességében 19 modulból áll, és a moduljai 1023 függvényt tartalmaznak. A függvény ágak száma 1453. A legtöbb függvény egy modulban 163, és a legtöbb függvény ág száma egy modulban 238. A modulokon mért ciklomatikus szám összege 3024, és ugyanennek a mértéknek az egy modulra mért legnagyobb értéke 704, ami kimagasló eredmény. (A forrásszöveget a dolgozatban nem közöljük, mivel az az Erlang disztribúciókban szerepel, és ezáltal szabadon hozzáférhető. Az eredmények a dolgozat írásakor elérhető verzióra vonatkoznak.) Ezek a számok megfelelővé teszik a szoftvert arra, hogy a transzformációs algoritmust tesztelhessük a segítségével.

A tesztek lefuttatása és a transzformációk hatáselemzése mellett a következő kérdésekre keressük a választ:

1. A modulok ciklomatikus számát a függvények ciklomatikus számának az összegével jellemezzük. Ez a számítási modell nem veszi figyelembe a függvények hívási

gráfját, ami torzítja a kapott értéket. Érdemes-e ezt az attribútumot vizsgálni a mérések során és hozzáadni azt az eredményhez?

2. Szintén a modulokkal kapcsolatban felvetődik az a kérdés, hogy melyik modul a bonyolultabb: Amelyik tíz darab, egyenként 1-es *McCabe* értékkel bíró függvényt tartalmaz, vagy az, amelyik egy darab 10-es *McCabe* értékkel rendelkezik?

3. A ciklomatikus szám minden függvény esetén legalább egy, mivel egy utat minimum tartalmaz. Ekkor, ha kiemeljük a mélyebben beágyazott szelekciós kifejezéseket az adott függvényből úgy, hogy ezzel új függvényt hozunk létre a kiemelt kifejezésből (lásd: 3 fejezetben), a modult jellemző ciklomatikus szám értéke indokolatlanul megnő (minden új függvény eggyel megnöveli). Ez minden újabb transzformációs lépést követően egyre jobban torzítja az eredményt. A kérdés ebben az esetben, hogy ezt a növekményt ki kell-e vonni a végeredményből?

4. Mindezeket figyelembe véve, a teljes modul ciklomatikus száma és a függvények egyesével mért ciklomatikus számának összege között mi az összefüggés?

5. Hogyan javíthatjuk az Erlang programok ciklomatikus számát a lehető legjobban és milyen átalakításokat kell eszközölni ahhoz, hogy a program lexikális felépítése, vagyis a programozói stílus is javuljon (közelebb álljon a funkcionális nyelek stílusjegyeihez)?

6. A McCabe féle ciklomatikus számot vesszük alapul az átalakítások során, elsőként elemezzük a mértéket.

   Porkoláb [34] cikkében utal rá, hogy a *McCabe*-féle ciklomatikus szám nem veszi figyelembe a vezérlő szerkezetek egymásba ágyazottságának a mértékét. Ez az érték viszont nem elhanyagolható, ha a tesztelés költségeit próbáljuk kikövetkeztetni a ciklomatikus szám alapján, de nem mellékes akkor sem, ha egyéb okokból mérjük a bonyolultságot.

   Minél jobban egymásba ágyazzuk a vezérlő szerkezeteket, annál nehezebb a programok megértése, átalakítása és tesztelése. Mindezeket figyelembe véve, figyeljük-e a program ezen attribútumait a mérések során?

7. Amennyiben az egyik függvény több, egymást követő szelekciós kifejezést tartalmaz, egy másik pedig ezeket egymásba ágyazza, egyenlőnek kell-e tekinteni a két függvény ciklomatikus számát?

A fejezetben kivitelezett mérések, és a hozzájuk kapcsolódó transzformációs lépéssorozatok elvégzésének elsődleges célja tehát az, hogy a vizsgált programszövegben található függvények ciklomatikus számán javítsunk, de mindezek mellet megvizsgáljuk néhány kapcsolódó mérték alakulását és ezek összefüggéseit is.

Elsőként a *Dialyzer* szoftver *dialyzer_dataflow* modulján futtatjuk le a transzformációs lépéseket, mivel ez az a modul, amelyben a legtöbb beágyazott *case* kifejezés található.

A transzformációs algoritmus futtatását megelőzően a modul forráskódot elemeztük a beolvasást végző elemző algoritmussal, majd az így kapott szemantikus gráfon (lásd: 2. fejezetben) meg kell mérnünk a modult jellemző *McCabe* bonyolultsági mértéket.

$$src \leftarrow analyzer(dialyzer\_dataflow)$$

Az első megoldási kísérletnél megmértük a függvények számát is (*number_of_fun*), majd vettük a két érték hányadosát: $\frac{mcCabe(src)}{number\_of\_fun(src)}$, ahol a $mcCabe(src)$ a forrásszövegen mért *mcCabe*-féle ciklomatikus szám, az *src* a mért forrásszöveg, és a *nof* a *number_of_function* bonyolultsági mérték, vagyis az összes modulban található függvény száma.

Az eredmény: $x_1 = \frac{704}{165} = 4.26666666667$. Ezt az értéket vesszük alapul, és a javítására irányuló szkripttel próbáljuk jó irányba változtatni, vagyis valamilyen módon javítani a modul ciklomatikus számán.

A ciklomatikus számot a teszt során azért osztjuk a függvények számával, mert amennyire lehet, ki kell küszöbölnünk a függvények számának növekedése miatt kialakult torzító hatást (a torzító hatások és ezek kiküszöbölésére még visszatérünk a fejezetben).

Kifejezések kiemelése

```
optimize
   extract_function (exprtype, case_expr)
where
   f_max_depth_of_cases > 1
limit
   1
```

4.16. ábra. Módosított szkript

Ha lefuttattuk azt a szkriptet a forrásszövegre, amelyet korábban, a 4.14 példában láthattunk és a 4.21 ábrán látható kiegészítésekkel láttunk el, (Ennek a pontos feladata, hogy az egy mélységnél mélyebben beágyazott *case* kifejezéseket kiemelje a *extract_function* transzformációs lépések segítségével, és a helyükre az általa létrehozott függvények hívását helyezze el. Ha a kiemelések során elsőként csak egy (*limit = 1*), majd két (*limit = 2*) szintjét emeljük ki a legmélyebben beágyazott kifejezéseknek.) a következő eredményt kapjuk:

$$script_1(src) = src'$$

$$x_2 = \frac{mcCabe(src')}{number\_of\_fun(src')} = \frac{794}{255} = 3,1137254901960785$$

$$\frac{x_2}{x_1} = 0.8473122889758293 = 84\% = 16\% \uparrow^{(limit=1)}$$

$$\frac{x_2}{x_1} = 0.729779411764706 = 72\% = 28\% \uparrow^{(limit=2)}$$

A jobb eredmény elérése érdekében megmérjük, hogy a modulban mi a *case* kifejezések maximális beágyazottsági szintje (*max_depth_of_cases*). A mérési eredmény hét szintet jelez, vagyis a szkript *limit* szakaszánál ezt az értéket kell megadnunk, ami arra utasítja a szkriptet, hogy legalább hétszer végezze el a kiemeléseket.

$$script_2(src') = src''$$

$$x_3 = \frac{mcCabe(src'')}{number\_of\_fun(src'')} = \frac{868}{329} = 2.6382978723404$$

$$\frac{x_3}{x_1} = 0.6183510638297872 = 61\% = 39\% \uparrow^{(limit=7)}$$

Az új eredményeket megvizsgálva levonhatunk néhány fontos következtetést.

Az első, hogy a ciklomatikus szám modulra mért értékei megnövekedtek az új függvények miatt. Összehasonlítva a függvények számát, és a ciklomatikus számot a transzformációk előtt és után, világosan látszik, hogy a $mcCabe(src) = mcCabe(src') - (nof(src) - nof(src'))$, vagyis a kiemelésekkel a ciklomatikus szám a modulra nézve nem változik meg abban az esetben, ha a beágyazottság mértékét és a függvények számát nem kalkuláljuk bele az értékbe.

Ez azért van így, mert a függvények kifejezéseiben szereplő "döntések" a modulban maradnak, vagyis attól, hogy egy függvény belsejében található döntést kiemelünk egy új függvénybe, az a modulból nem tűnik el (ezért osztottuk korábban az értéket a függvények számával).

A modulra mért értékek mellett figyelembe kell vennünk a modulban szereplő függvények egyesével mért ciklomatikus számát, valamint a maximumot és a minimumot ezek közül. Ha ezen eredmények változásait összehasonlítjuk a transzformációk előtt és azokat követően, csak ebben az esetben kaphatunk pontos képet a transzformáció hatásairól. (Az átlagos értékek a modulra különben sem nevezhetőek pontosnak, és az eredetileg a modulban lévő függvények száma is nagyban befolyásolhatja az eredményt, mivel minden új függvény hozzáad legalább egyet az értékhez...)

Az elvégzett méréseket vizsgálva láthatjuk, hogy a transzformációk előtt mért ciklomatikus szám összege, amennyiben nem osztunk a függvények számával, a transzformációk előtt 704, azokat követően 794, a függvények száma a transzformáció előtt 165, azt követően 225. A $794 - 704 = 255 - 165$, vagyis az újonnan behozott függvények hozzák magukkal az érték növekedését.

Mindezek fényében azt a javaslatot tehetjük, hogy a *McCabe*-féle ciklomatikus szám mérésekor ne a modulra mért összeget vegyük figyelembe, hanem a modul függvényein mért legnagyobb értéket, és ezt hasonlítsuk össze a transzformáció elvégzése előtt és azt követően.

$$max(mcCabe_f(src)) > max(mcCabe_f(src'))$$

Vegyük továbbá figyelembe különböző vezérlő szerkezetek egymásba ágyazottságának a mértékét, így a következő összefüggés alapján kell számoljunk, és ez alapján kell kidolgoznunk a megfelelő transzformációs szkriptet is. Az eredmény kiszámítása a kiindulási forrásszövegre a következő:

$$mcCabe(src) + sum(max\_dept\_of\_struct(src)) - z$$

$$+number\_of\_exceptions(src)$$

A beágyazottságok maximumából kivonhatjuk azoknak a függvényeknek a számát, ahol a beágyazottság mértéke egy (vagy az az érték, amire a szkriptben optimalizálunk), mivel ez is torzíthatja az értéket (a képletben ezt az értéket z-vel jelöljük). A $+(number\_of\_exceptions)$ rész, amely a kivételkezelők által behozott döntéseket számolja opcionális, de ha a kiindulási állapotra használjuk, akkor a transzformáció utáni állapot kiszámításánál sem hagyhatjuk ki. (Még pontosabb eredményekhez jutnánk, ha a kivételkezelők ágait, vagyis a kivétel lehetséges kimeneteleit is hozzávennénk az eredményhez.

Ezen a ponton bevezetünk egy újabb mértéket, de csak a jobb eredmény elérése érdekében. Ez a mérték a programokban található kivételkezelők számát adja vissza modul és függvény típusú csomópontokra. A mérték implementálásához a függvény kifejezések számát mérő mértéket megvalósító függvényt alakítjuk át úgy, hogy az ne csak a kifejezések (fun_expr), hanem a kivételkezelők (try_expr) számát is vissza tudja adni.

Az Erlang nyelvben a kivételkezelő try blokk tartalmazhat a *case* vezérlő szerkezetekben megszokott, mintaillesztésekre épülő ágakat, és a *catch* blokkban is többfelé ágazhat a program vezérlés. A megoldás tehát nem a kivételkezelőkben található döntések, hanem csak a kivételkezelők számát adja vissza, így nem teljesen pontos, de

a célnak megfelel.) A transzformált szövegre az alábbi formulával számítható ki az eredmény:

$$mcCabe(src') + sum(max\_dept\_of\_struct(src')) - z$$
$$+number\_of\_exceptions(src')$$

Ezután a függvényekre mért maximum értékéből, valamint a képlettel kiszámolt értékekből már nagy biztonsággal el lehet dönteni, hogy az eredmény jobb-e, mint a kiindulási érték. A számítási módszer már figyelembe veszi a beágyazottsági mélységet, minden szinttel eggyel növeli az adott függvény/modul ciklomatikus számát. Az Erlang programok tartalmaznak kivételkezelő rutinokat is, amelyek újabb végrehajtási útvonalakat eredményeznek, vagyis növelik a ciklomatikus számot. Ezeket a kifejezéseket szintén hozzászámolhatjuk a kiindulási, és a transzformációkat követően kapott eredményekhez. Sajnos ez a módszer a kiegészítésekkel együtt sem tökéletes, mivel a modulra mért értékek tekintetében nem veszi figyelembe a függvények közti kapcsolatokat és az így kialakuló döntési helyzeteket, ami a hívási gráf alapján feltérképezhető, de mindenképpen jobb a korábbiaknál.

Ahhoz, hogy az előzőnél reprezentatívabb eredményt kapjunk, a *Dialyzer* szoftver teljes forráskódját elemeztetnünk kellett a forráskód beolvasását végző algoritmussal, majd ezt transzformálnunk. A lépéssorozat elvégzéséhez az előzőekben alkalmazott szkriptet használtuk fel, viszont figyelembe vettük a javasolt módosításokat, vagyis a beágyazottságot hozzáadtuk az eredményhez, és a függvényekre mért minimumokat és maximumokat is vizsgáltuk, mikor levontuk a következtetéseinket.

A ciklomatikus szám függvényekre mért értékének maximuma az átalakítás előtt $max(mcCabe_f(src)) = 96$, az átalakítást követően $max(mcCabe_f(src')) = 73$, vagyis a $max(mcCabe_f(src)) > max(mcCabe_f(src'))$.

Az eredmények javulást mutatnak, de a szkript minden modul minden olyan függvényében elvégzi a kiemeléseket, amelyek egynél nagyobb beágyazottsággal rendelkeznek. Ez a beágyazottsági mélység viszont nem feltétlenül rossz. Lehetőség szerint csak azokon a helyeken kellene kiemeléseket végezni, ahol erre feltétlenül szükség van, vagyis azokban a modulokban, ahol a függvényekre mért ciklomatikus számok maximuma nagy.

Ezeket a kiegészítéseket szem előtt tartva, és a javuló eredményeket látva a 4.21 ábrán látható transzformációs szkriptet javasoljuk a szoftver ciklomatikus számának javítására.

A szkript a kimagasló ciklomatikus számmal rendelkező modulokban kiemeléseket végez abban az esetben, ha az adott függvény beágyazottsága nagyobb az elvárt értéktől. A *limit* értékét aszerint kell módosítani, hogy mekkora annak a legnagyobb beágyazottság mélysége. Erre a lépésre azért van szükség, hogy minden beágyazott ki-

Mérték	Átalakítás előtt	Átalakítás után
mcCabe module	3024	3530
mcCabe module min	2	2
mcCabe module max	704	878
mcCabe function	3024	3530
mcCabe function max	96	73
number_of_fun module	1023	1529
number_of_fun module min	2	2
number_of_fun module max	165	339
max_depth_of_cases module max	7	1
number_of_funclauses module	1453	1959
number_of_funclauses module max	238	391
max_depth_of_struct module	71	28
max_depth_of_struct module max	8	4
number_of_exceptions module	42	42
number_of_exceptions module max	10	10
mcCabe/number_of_fun	2.95601173	2.308698496

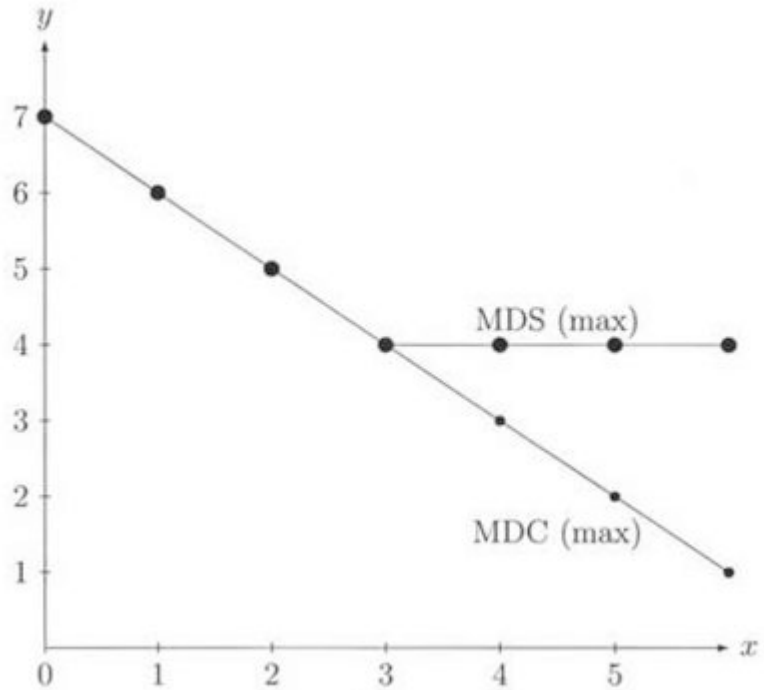
4.1. táblázat. Az eredeti és a javított kódon mért értékek

fejezésre sor kerüljön. A kiemeléseket a reprezentatív eredmény elérése érdekében úgy végeztük el, hogy első lépésben csak azokra a beágyazott *case* kifejezésekre alkalmaztuk a transzformációkat, amelyek bágyazottsága nagyobb, mint hat. Ezután minden lépésben csökkentettük ezt az értéket eggyel mindaddig, amíg az egy mélységben elhelyezkedő kifejezések kiemelése meg nem történt.

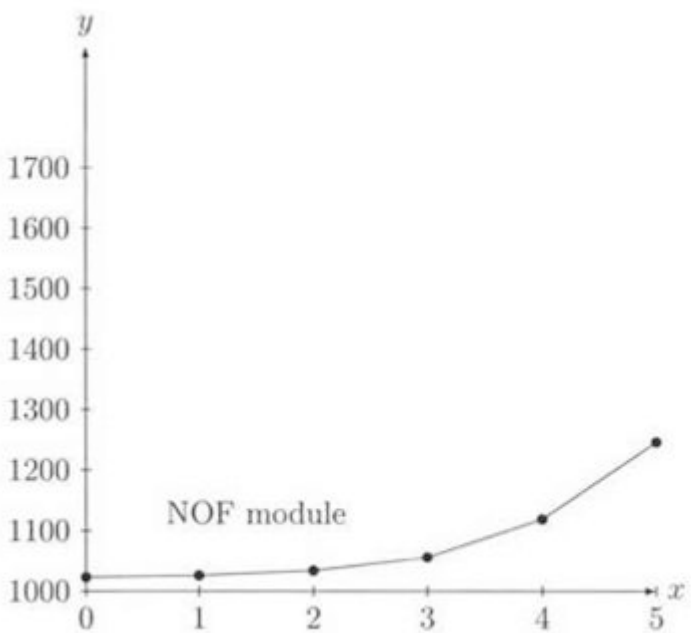
Minden lépést követően megmértük azokat a mértékeket, amelyeket az előzőekben említettünk, a kifejezések beágyazottságát a modulokra, a *McCabe* szám minimumát, maximumát és összegét a függvényekre és a modulokra, valamint a függvények számának alakulását a modulokra nézve. A kapott eredményeket a 4.17, 4.18, 4.19, 4.20 diagramokon láthatjuk.

Amennyiben a szkript által produkált eredményen tovább szeretnénk finomítani, a *max_depth_of_cases* helyett a *max_depth_of_structs* mérőszámot alkalmazhatjuk a feltételben.

Viszont a mérték alkalmazása mellett kerülnünk kell azokat a helyzeteket, mikor az általános beágyazottsági szint nagyobb mint kettő, de a *case* kifejezések beágyazottsága egy, vagy nulla. Erre azért van szükség, hogy ne emeljük ki azokat az elágazásokat, amelyek a legfelső szinten helyezkednek el. Ha így tennénk, a szkript addig végezné a kiemeléseket, amíg a *limit* értéke meg nem állítaná a futását. Erre az esetre bevezethe-



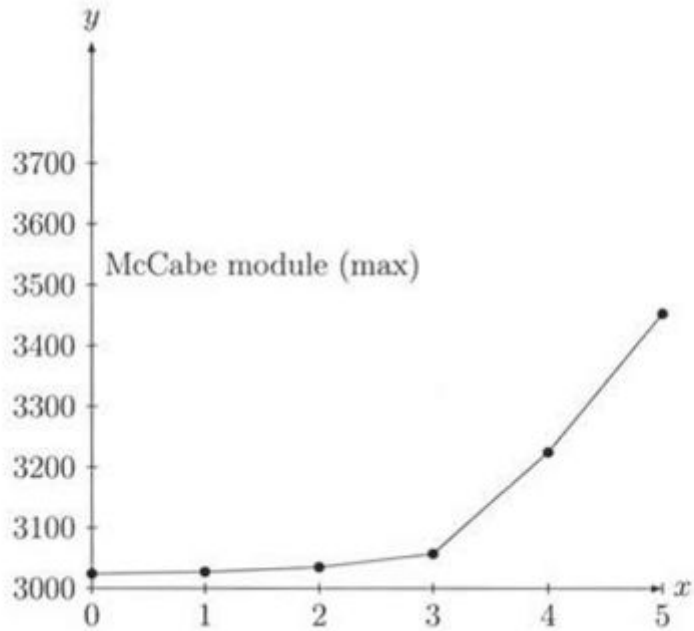
4.17. ábra. A struktúrák (MDS) és a *case* kifejezések (MDC) beágyazottságának alakulása a transzformációs lépések során



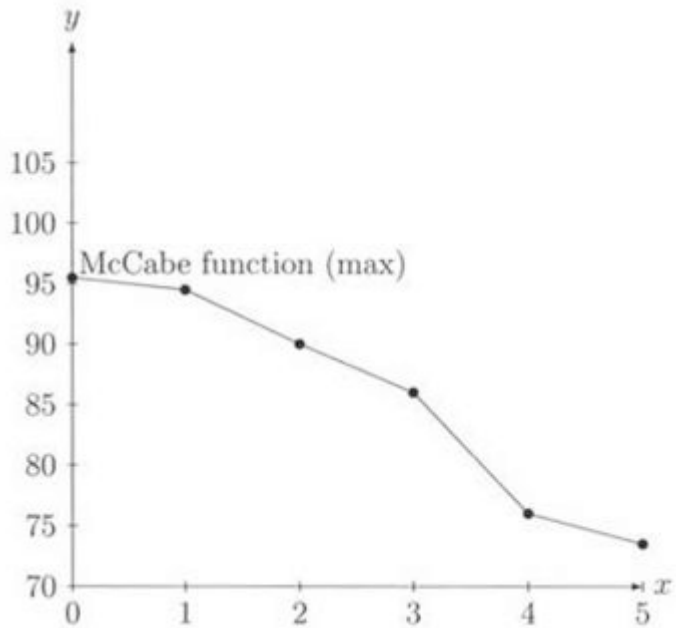
4.18. ábra. A modulokban található függvények számának növekedése a transzformációkat követően

tünk egy olyan feltételt, amely az általános beágyazottság mértéke alapján csak akkor végzi el a kiemeléseket, ha a *case*-ek is egymásba vannak ágyazva legalább egyszer.

Más részről a függvények száma nagyon megnövekedhet a szkript alkalmazása során, és ez szintén a komplexitást érintő problémává válhat. Egy lehetséges megoldása a helyzetnek, ha a kiemelt kifejezéseket tartalmazó függvényekre úgy tekintünk, mint segéd



4.19. ábra. A modulokra mért McCabe szám összegének alakulása a transzformációk során



4.20. ábra. A függvények McCabe számának maximuma az egyes transzformációs lépéseket követően

függvényekre, és átmozgatjuk őket egy új modulba, amit láthatóvá teszünk a forrás modulokban. Ez a szkript a 4.22 ábrán látható. (A *like* kulcsszó után az "opt_" előtagot keressük, mivel ez az alapértelmezett kezdőtagja a generált függvényneveknek.) Ennek a műveletsornak az elvégzése nem ütközik akadályokba, és a program jelentését sem befolyásolja, mivel az átalakítások során létrejött függvényekre nincs más függvényekből jövő hívás, és a kiemelések hatása ilyen szempontból teljesen lokális a kifejezéseket

McCabe szám

```
optimize
   extract_function (exprtype, case_expr)
where
   f_mcCabe > 6
    and
   f_max_depth_of_structures > 2
    and
   f_max_depth_of_cases > 1
limit
   7;
```

4.21. ábra. Javított kiemelést végző szkript

Új függvények kiküszöbölése

```
optimize
  move_fun (targetmod, storage)
where
  f_name like 'opt'
limit 1
```

4.22. ábra. Függvények mozgatása

eredetileg tartalmazó függvényekre. Azt is megtehetjük, hogy egyszerűen nem vesszük figyelembe a kiemelt kifejezéseket tartalmazó függvényeket a mérések során.

Az első esetben az új modul és a régi kapcsolata, vagyis a kötés közöttük növekszik, és számos kompenzációs lépés végrehajtása szükségessé válhat (rekordok mozgatása, makrók láthatóvá tétele, függvények importja, stb...), és ez szintén növelheti a komplexitást, de az eredetileg kitűzött célban megfogalmazottaknak nem mond ellent, vagyis a kiindulási modul függvényeinek ciklomatikus száma jobb eredményt fog mutatni ennek a néhány soros szkriptnek a lefuttatását követően.

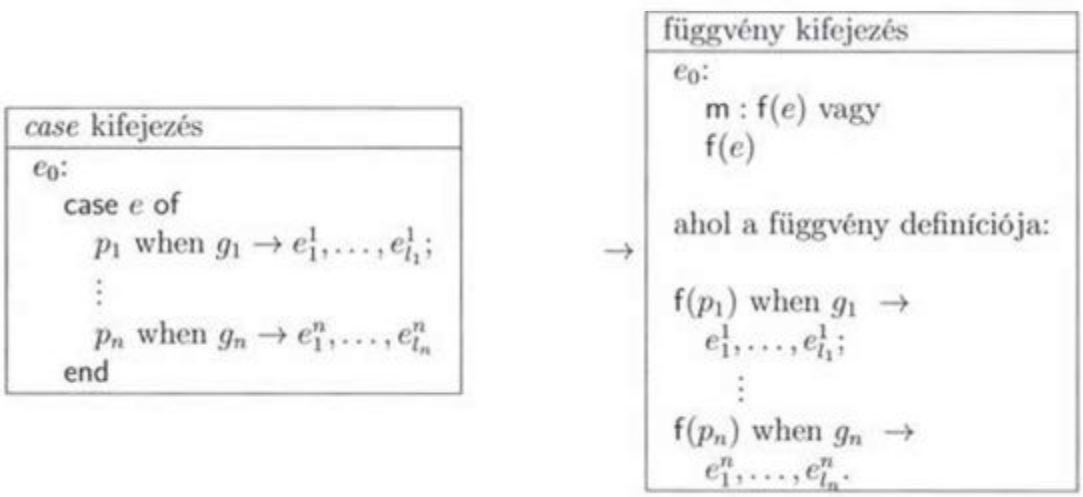
Az itt bemutatott lépések mintájára, a szkriptnyelv felhasználásával készíthetünk más mértékeket, vagy egyszerre több mértéket mérő és azokat javító szkripteket, de megtehetjük azt is, hogy a mértékek vizsgálatával és a kapott értékek alapján vezérelt transzformációkkal a programozói stílust alakítjuk át.

4.5.1. Programozói stílus javítása

A *case* kifejezések kiemelése javítja a forrásszöveg bonyolultságát, de a *case* kifejezéseket nem szünteti meg. Ahhoz tehát, hogy a bemutatottnál talán nem jobb, de "szebb" eredményt érjünk el, a *case* kifejezéseket át tudjuk alakítani függvény ágakká a 4.23

példában látható módon (az ábrát - és a hozzá tartozó transzformáció hatáselemzését - a 3. fejezetben már bemutattuk, itt csak ismételjük).

Az Erlang nyelvre és az általunk vizsgált programokat nézve az Erlang programozókra jellemző a *case* kifejezések túlzott használata, de a funkcionális nyelvnek inkább sajátja a függvény ágak, vagy ismertebb nevükön az *overload* típusú függvények alkalmazása.



4.23. ábra. Case kifejezés kiemelése

A 4.23 ábrán az $e_1, ..., e_n \in E$ Erlang kifejezések, $p_i \in P$ Erlang minták, és $g_i \in G$ őr feltételek, melyek a függvények és a *case* kifejezés ágaiban fordulhatnak elő.

Az itt bemutatott transzformációs lépés elvégzését követően a kiemelt *case* kifejezéseket eredetileg tartalmazó függvények ciklomatikus száma csökken, de az új függvények ciklomatikus száma növeli az összeget.

A transzformációt követően nem kapunk lényegi javulást a ciklomatikus szám tekintetében, mivel a függvény ágak számát ugyanolyan döntésnek kell tekintenünk, mint a szelekciót (ezt az algoritmus eredetileg is beleszámolta az eredménybe). A bonyolultság viszont összességében javuló tendenciát mutat, ha elfogadjuk, hogy a funkcionális nyelvekben, így az Erlang nyelvben is a függvény ágak használata az inkább követendő példa, nem a beágyazott kifejezések, vagy az elágazások használata, mivel alkalmazásuk által áttekinthetőbbé válik a forrásszöveg. (Ez megítélés kérdése. Amennyiben az elágazásokat tartjuk jobbnak, transzformálhatjuk ebbe az "irányba" is a programszöveget).

4.6. Összefoglalás

Az *Automatikus programtranszformációk* című fejezetben bemutattuk a 2.2.5. fejezetben kidolgozott lekérdező nyelv kiterjesztését. Az így kapott nyelv segítségével most már nem csak lekérdezések, hanem bonyolultsági mértékek mérésén alapuló, automatikus programtranszformációk leírására is lehetőségünk nyílik.

Definiáltuk a szoftverbonyolultsági mértékeken alapuló automatikus transzformációs lépéssorozatok leírására és futtatására alkalmas nyelv szintaxisát, és ismertettük a nyelvhez konstruált elemző és futtatást végző algoritmus működési elvét.

A szintaxis és használati esetek ismertetése mellett a fejezet bemutatja a nyelv alkalmazásához szükséges algoritmust, illetve azt, hogy milyen eredményeket érhetünk el a használatával.

A transzformációs nyelv szakaszokból épül fel és minden szakasz három részből áll. Az első, *optimize* kulcsszó után írt részben azt a transzformációt, és a transzformáció paramétereit adhatjuk meg, amelyet az adott lépés elvégzésére alkalmazni kívánunk.

A második, *where* címkéjű részben bonyolultsági mértéken alapuló feltételeket definiálhatunk, amelyek alapján kiválasztásra kerülnek a transzformációs folyamatban résztvevő szemantikus gráf csomópontok (lásd: a 2. fejezetben). A feltételek vonatkozhatnak függvény és modul típusú csomópontokra. Ez a rész indítja el az algoritmusba épített függvényeket, amelyek elemzik a teljes forrásszöveget, kiválasztják a feltételeknek megfelelő gráf csúcsokat, majd a kiválasztott csúcsok és a transzformáció típusa alapján előállítják az *optimize* részben ismertetett, és az adott lépés elvégzéséhez szükséges paramétereket, továbbá a paraméterek összegyűjtése mellett a szabályrendszer alapján megállítják a szkriptek futását.

A harmadik, *limit* kulcsszóval bevezetett rész szabályozza a szkriptek futását úgy, hogy az itt megadott érték alapján korlátozza a lépésszámot. Erre akkor van szükség, ha egyébként a *where* feltételrendszerben megadott szabályok nem állítják meg az átalakítást végző folyamatot.

A fejezet második felében a transzformációs nyelv szintaxisát, a nyelv megkonstruálásának problémáit, különböző programkonstrukcióit és egymástól jól elkülöníthető szekcióit, végül példaprogramokon keresztül a használatában rejlő lehetőségeket ismerhettük meg.

Mindezeken kívül elkészítettünk számos, a bonyolultsági mértékeken alapuló, forráskód átalakítási problémát megoldó szkriptet, majd megmutattuk a futási eredményeiket, igazolva ezzel a használhatóságukat.

4.7. A tézis megfogalmazása

Forráskód automatikus transzformációja szoftverbonyolultsági mértékek alapján, és kódminőség javító transzformációs sémák kidolgozása.

Létrehoztunk az Erlang programok bonyolultsági mértékeit mérő, és a mértékeken alapuló automatikus transzformációkat végző algoritmust. Definiáltuk azt a szkript nyelvet, amely lehetővé teszi az Erlang nyelvű program konstrukciók bonyolultságának mérését, valamint a kapott eredmények alapján történő átalakítását szolgáló transzformációs lépések leírását.

4.8. Releváns publikációk

- Roland Király. *Results of complexity metric based automatic source code transformations.* Annales Mathematicae et Informaticae 42 (2013) accepted
- Roland Király, Róbert Kitlei. *Metric Based optimalization of functional source code.* Annales Mathematicae et Informaticae 37 (2011) Pages: 59-74
- Király, R., Kitlei R.: *Complexity measurments for functional programming languages* (MaCS 2010) Studia Universitatis Babes-Bolyai Series Informatica ISSN: 1224-869x (paper version) ISSN: 2065-9601 (online version)
- Király, R., Kitlei R.: *Complexity measurments for functional code* 8th Joint Conference on Mathematics and Computer Science (MaCS 2010) refereed, and the proceedings will have ISBN classification July 14-17, 2010

5. fejezet

Összefoglalás

A dolgozatban bemutattuk azt a bonyolultsági mértékeken alapuló elemző, és optimalizáló algoritmust, amelyet *Erlang* nyelvű forrásszövegek készítése során, valamint korábban készült, de átalakításra váró programszövegek automatikus, vagy fél-automatikus javítására használhatunk.

A *Erlang forrásszöveg gráf reprezentációja bonyolultság méréséhez* című fejezetben bevezettük az $\mathcal{SG}$ gráfot a forrásszöveg bonyolultságának méréséhez és tárolásához. Ez az adatszerkezet a bonyolultságot elemző algoritmus működéséhez szükséges.

A *Erlang programok bonyolultsági mérőszámai* című fejezetben bemutattuk azokat a strukturális bonyolultsági mértéket, amelyeket az Erlang forrásszövegek bonyolultságának méréséhez állítottunk össze már meglévő mértékek átalakításával és újak kidolgozásával.

A *Szöveges lekérdező nyelv* című fejezetben ismertettük a bonyolultsági mértékek kiszámításához kidolgozott strukturált lekérdező nyelvet, amely lehetővé teszi a bonyolultsági mérték mérését a bonyolultságot elemző algoritmus ismerete nélkül.

Ebben a három fejezetben az ***Erlang nyelvre alkalmazható metrikák kidolgozása, mérése és lekérdező nyelv megalkotása*** tézis eredményeit ismertettük.

A *bonyolult programrészek lokalizálása* című fejezetben bemutattuk a módszert, amellyel a szoftverbonyolultsági mértékek eredményei alapján lokalizálni tudjuk a kezelhetetlenül bonyolult programrészeket, és megvizsgáltuk, hogy a különböző programtranszformációk hogyan hatnak a kódminőségre.

A fejezet a ***Bonyolult programrészek lokalizálása és kódjavítást célzó programtranszformációk hatása a kódminőségre*** című tézist és annak eredményeit ismerteti.

Az *Automatikus program transzformációk* című fejezetben megvizsgáltuk annak a lehetőségét, hogy a bonyolultsági mértékek mérésével és elemzésével hogyan lehet automatizált javítását célzó programtranszformációkat megvalósítani.

A fejezet a ***Forráskód automatikus transzformációja szoftverbonyolultsági mértékek alapján és kódminőség javító transzformációs sémák kidolgozása*** című tézis eredményeit mutatta be.

A transzformációkhoz kidolgozott nyelv segítségével olyan szkripteket készíthetünk, amelyekkel a fejlesztés alatt álló, vagy korábban már elkészített, de kezelhetetlen méretű forrásszöveg bonyolultsági mértékek alapján detektálható hibái automatikusan javíthatóvá válnak. A fejezet végén példaprogramokkal igazoltuk az automatikus hibajavítás működőképességét.

A dolgozatba helyet kaptak azok a mérési eredmények is, amelyekre az implementációs lépések kidolgozását alapoztuk, illetve a fejezetekben megtalálhatjuk azokat a bonyolultsági mértékeken alapuló, a transzformációs nyelv segítségével definiált optimalizációs szkripteket, amelyek felhasználásával számos Erlang programot transzformáltunk annak érdekében, hogy igazoljuk az algoritmusok és a módszerek működőképességét.

Köszönetnyilvánítás

A munkám során mind a kutatási fázist, mind az implementációs és tesztelési munkákat tekintve nélkülözhetetlen segítséget kaptam az ELTE Informatika Karának oktatóitól, professzoraitól, valamint az ELTE IK és az Ericsson Magyarország támogatásával működő Erlang programok rerfaktorálásával foglalkozó kutató csoport tagjaitól, akik fáradságot nem ismerve minden - megoldhatatlannak tűnő - probléma támasztotta akadályon átsegítettek.

Külön köszönet illeti témavezetőmet, Dr. Horváth Zoltánt, Tóth Melindát, Lövei Lászlót, Kitlei Róbertet, Dr. Kozsik Tamást, és Dr. Tejfel Mátét, valamint a kutatócsoport minden tagját. Mindezeken kívül köszönöm a sok segítséget Dr. Hernyák Zoltánnak, Dr. Kovács Emődnek, Dr. Liptai Kálmánnak, és minden kollégámnak.

Végül nagyon köszönöm a türelmet és a féltő támogatást Feleségemnek és három gyönyörű gyermekemnek!

6. fejezet

Mellékletek

6.1. A dolgozat rövid összefoglalója

A kutatási témánk az Erlang programok bonyolultságának a mérése és a szoftver bonyolultsági mértékek alapján történő forráskód minőség javítás.

A kutató munka során bemutattuk azt a szemantikus gráfot, amely a forrásszöveg bonyolultságának a méréséhez került kidolgozásra. Ismertettük ennek az adatszerkezetnek a továbbfejlesztett változatát, amely alkalmas a bonyolultság mérésére és tárolására, valamint tartalmazza az adott program minden, számunkra fontos, statikus analízissel mérhető attribútumát.

Bemutatásra kerültek azok a strukturális bonyolultsági mértékek, amelyek az Erlang forrásszöveg bonyolultságát jól jellemzik, valamint a bonyolultsági mértékek méréséhez kidolgozott strukturált lekérdező nyelv, amely használata lehetővé teszi tetszőleges bonyolultsági mérték mérését a programszövegen a bonyolultságot elemző algoritmus ismerete nélkül. Bemutattuk a nyelv szintaxisát, valamint példákon keresztül a használatában rejlő lehetőségeket.

Ismertettük a szoftver bonyolultság alapú hibadetektáló algoritmus működési elvét, és az elemzések elvégzésére kidolgozott szabályrendszert, ami segít a programok fejlesztése közben, vagy a programkódok utólagos átalakítása során, hogy könnyen felismerhessük azokat a programfejlesztési hibákat, amelyeket a bonyolultsági mértékek változásai alapján detektálni lehet.

Megvizsgáltuk annak a lehetőségét, hogy a bonyolultsági mértékek mérésével, valamint az elemző és hibadetektáló algoritmus alkalmazásával hogyan lehet a bonyolultsági mértékek automatizált javítását célzó program transzformációkat megvalósítani.

Bemutattuk az automatizált transzformációs lépések gyakorlati alkalmazásait is és kiegészítettük a lekérdező nyelvet olyan elemekkel, amelyek a bonyolultság mérése mellett segítik a bonyolultsági mérték alapú, automatikus transzformációk megírását.

Az így kapott transzformációs nyelvtan lehetővé teszi olyan szkriptek írását, amelyekkel a fejlesztés alatt álló, vagy korábban már elkészített, de kézzel javíthatatlan méretű forrásszöveg bonyolultsági mértékek alapján detektálható hibái automatikusan javíthatóvá válnak. Példaprogramok, és futási eredményeik bemutatásával igazoltuk az automatikus hibajavítás gyakorlatban való alkalmazhatóságát.

A kutatás során bemutatásra kerülő algoritmusokat implementáltuk és lehetővé tettük azok prototípus jellegű, ipari alapú felhasználását. Az eredményeket publikáltuk számos folyóiratban és konferencián.

6.2. Short Summary

Our research topic is the measurement of the complexity of Erlang programs and the improvement of the quality of the source code based on software complexity levels.

The semantic graph was presented in this research, which was developed for the measurement of the complexity of the source code. We also described an improved version of the data structure, which is suitable for measuring and storing the complexity, and includes all of the program's attributes that are relevant to us, that can be measured with static analysis.

We presented those structural complexity levels that characterize the complexity of the Erlang source code well; alongside the structured query language developed to measure the complexity levels, which allows the measurement of arbitrary complexity levels in the program code without knowing the complexity parsing algorithm. We introduced the language's syntax and through examples the potential opportunities of its usage.

We discussed the operating principle of the error detecting algorithm which is based on software complexity, and the set of rules developed to perform the analyzes, that help during the development of the programs or during the subsequent conversion of the existing code, to easily recognize those software development errors, which can be detected based on variations in the levels of complexity.

We examined the possibility of implementing an automated transformation program to improve the complexity measures, with the help of the evaluation of the complexity measures and with the use of an analyzing and error detecting algorithm.

We also showed the practical implementation of the automated transformation steps and supplemented the query language with elements that in addition to the measurement of the complexity measures assist in the writing of automated transformations based on complexity.

The resulting transformation meta-grammar allows the writing of scripts, which automatically improve, based on detectable complexity level errors, the source codes that are manually unrepairable in under development or finished programs.

The algorithms presented in the research were implemented and their prototype-based industrial use was made possible. The results are published in numerous journals and conferences.

6.3. A transzformációs algoritmus implementált változatának Emacs integrációja

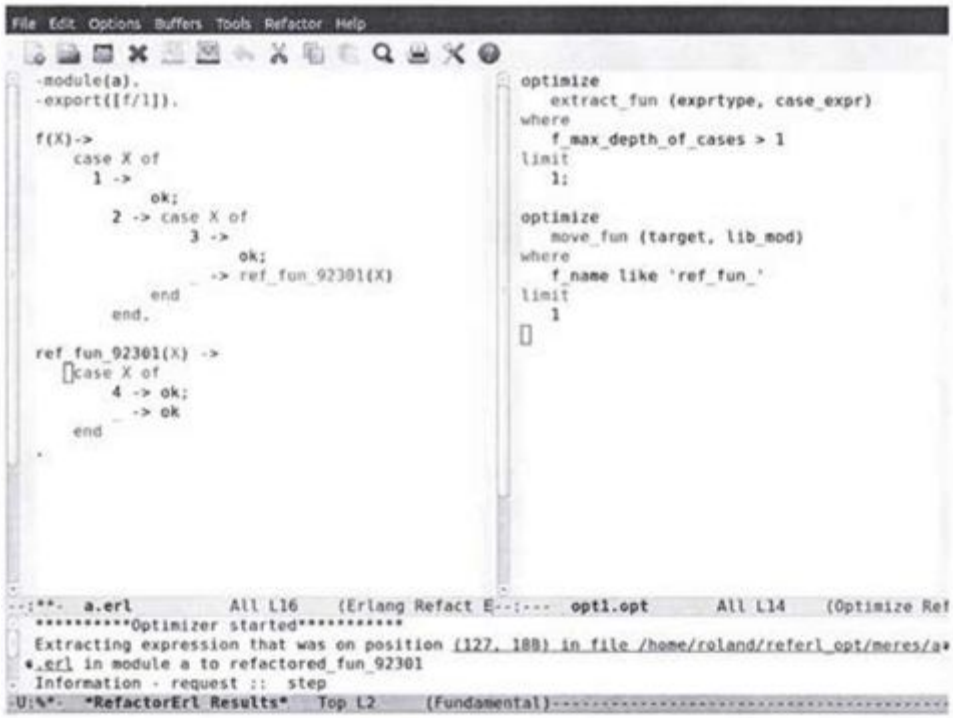
6.1. ábra. A hibakereső algoritmus működés közben

6.4. Az ismertetett bonyolultsági mértékek validálása

Miután megismertük az elemző algoritmusban használt bonyolultsági mértékeket, felmerülhet a kérdés, hogy a mértékeket implementáló függvények valóban jól mérik-e a forrásszöveg különböző, bonyolultsággal kapcsolatos tulajdonságait.

Azért, hogy erről meggyőződjünk, a fejezetben bemutatott bonyolultsági mértékeket Erlang forrásszöveg mérésével teszteltük. A tesztelés célja az volt, hogy bizonyítsuk a kiszámítást végző függvények helyes működését. A bizonyításhoz három különböző módszert is alkalmaztunk, amelyeket az alábbi felsorolásban láthatunk:

- Az első, hogy az általunk használt bonyolultsági mérték mindegyikéhez készítettünk egy rövid és egyszerű modult. Az így elkészített modulok mérése során



6.2. ábra. A transzformációs szkript futtató környezetének Emacs integrációja

kapott triviális eredményeket fejben kiszámítva is lehetett ellenőrizni. Ez a fajta mérés jó eredményre vezetett, de lassú és nem használható nagy méretű forrásszöveg mérésére.

- A második módszer lényege, hogy készítettünk egy automatikus forráskód generátort, amely minden mértékhez generál olyan Erlang modulokat (lásd: 6.4), amelyek az adott mértékre jellemző programnyelvi konstrukciókat tartalmazzák egyre nagyobb számban, vagy bonyolultságban. Az így előállított modulokban előre beállított számú, vagy bonyolultságú programelem található. A beállításokat definiáló paramétereket a mérések során kapott eredményekkel össze lehet hasonlítani. Ez a módszer is jó eredményeket adott.
- A harmadik módszer az első kettőnél jóval bonyolultabb. Itt egy viszonylag nagy és összetett szoftver kódját vizsgáltuk, majd elemeztük a kapott eredményeket. Olyan szoftvert kellett választanunk, ahol a modulok és a bennük található forrásszöveg mellett ismertek voltak az egyes átalakításokat indukáló célok és elvárt eredmények is. Erre azért volt szükség, hogy a kapott eredményekről el tudjuk dönteni, hogy azok összhangban vannak-e fejlesztés során elvárt eredményekkel. A *RefactorErl* forrásszövegét választottuk, és a fejlesztés két egymástól jól elkülöníthető fázisaiban megmértük, majd elemeztük a kapott eredményeket.

Az első fázis méréseit követően a *refaktoráló* eszköz egy komolyabb átalakításon esett át, amely során annak funkcionalitása jelentős mértékben kibővült, és a korábbi funkciók csoportokba (modulokba) lettek rendezve. Az átalakítást a szoftver bonyo-

```
optimize
   extract_fun (exprtype, case_expr)
where
  f_mcCabe > 10
   and
  f_max_depth_of_structures > 2
   and
  f_max_depth_of_cases > 1
limit 5;

optimize
   move_fun (targetmod, 'funlib')
where
  f_external_calls > 1,
  f_internal_calls < 1,
  f_external_calls_from > 1,
  f_internal_calls_from < 1,
  m_number_of_fun > 1
limit 2;

optimize
   rename_fun (prefix, 'lib_')
where
  module == 'funlib'
limit 5;

optimize
  tuple_funpar
where
  f_number_of_vars > 1
limit 1;

optimize
  introduce_record
where
  m_name == 'funlib',
  f_number_of_vars > 0
limit 1
```

6.3. ábra. Modulszervező szkript

lultságának változása és ezzel párhuzamosan a modulok (kezelhetetlenül) nagy mérete tette szükségessé. Mindezek mellett az egymással függvényhíváson keresztül és egyéb módon kapcsolódó, összetartozó függvények csoportosításának igénye is felmerült.

Generált modul mérések ellenőrzéséhez

```
-module(mod01).
-export([f01/1, f02/1, f03/3]).

%%% gencode
%%% max_depth_of_cases
%%% number_of_functions:4
%%% nested_expressions(case):0-4
%%% number_of_funparams:1
%%% returnn_value:simple_input_param_first

f01(P01) -> P01.

f02(P02) ->
    case P02 of
        0 -> P02;
        1 -> P02;
        _ -> P02
    end.

f03(P03) ->
    case P03 of
        0 -> P03;
        1 -> P03;
          _ ->
        case P03 of
            3 -> P03;
            4 -> P03;
            _ ->
        end
    end.
...
```

6.4. ábra. Generált program a beágyazottság és a függvényszám mérésére

6.1. megjegyzés. *A 6.1 táblázatban összegezve láthatjuk a mérési eredményeket. A kettő, vagy több értéket tartalmazó cellák több "magas", vagy éppen alacsony értéket is megmutatnak, mivel ezeken a pontokon a forráson mért értékek túlságosan elmozdultak az átlaghoz képest, és ezáltal nehezen értékelhető az eltérés.*

Az átalakítás során a programszöveg egymástól jól elkülönített, interfészeken keresztül kommunikáló rétegekre bontása lehetővé tette a program továbbfejlesztését. Az átalakítást követően a modulok közötti kapcsolat lazábbá vált, de a modulokon belüli kapcsolatok megerősödtek a függvények csoportosítása miatt. A mérések is ezt igazolták. A *coupling* modulokra mért átlagos értéke 19.41-ről 18.35944-re csökkent,

míg az összege 932-ről 4968-ra növekedett. Az Erlang forrásszöveg elemzésére generált elemző programot készítettünk, és a generátor által előállított modulokat használtuk fel a programban. A modulok közti összetartó erő ezáltal megnőtt, mert a generált elemző két modulja között rengeteg függvényhívás szerepel, és a különböző "*library*" modulok függvényeit is szinte minden más modul használja.

A modulok egyes csoportjai nagyon sok kapcsolaton keresztül, míg más modulok csak nagyon kevés híváson keresztül kapcsolódnak egymáshoz. A modulok közt mért összetartó erő átlaga csökkent, mivel a generált, és a "*library*" modulokon kívül a többi modul nem, vagy csak kis számban hívja egymás függvényeit (az átalakítások egyik célja éppen ennek az elérése volt).

A kohézió, vagyis a modulokon belüli "összetartó erő" szintén megnőtt (az átlaga is), mivel a modulokon belül bonyolódik a legtöbb függvényhívás (kivéve persze az imént említett "library" modulok függvényeire történő hivatkozásokat).

Mérték	Átalakítás előtt	Átalakítás után
Coupling	932	4968
Coupling (avg)	19.416666	18.35944
Cohesion (avg)	26.08	38.27649
Cohesion (max)	231/79/78/70	2548/680/329/164
Effective lines of code (sum)	14518	32366
Effective lines of code (avg)	308.89	425.868
Effective lines of code (max)	812/701/745	8041/1022/770
Number of functions (sum)	1329	2648
Number of functions (avg)	17.038	21.7
Number of functions (max)	85/70/49	551/91/64
Max depth of cases (avg)	1.7	1.6
Max depth of cases (max)	4	4
Branches of recursion (sum)	201	750
Branches of recursion (avg)	6.931	18.75
Branches of recursion (max)	20/18/15	211/43/35
Num. of function clauses (sum)	1725	6778
Num. of function clauses (avg)	36.70	89.18
Num. of function clauses (max)	139/133/53	3251/303/165
Number of fun-expression (sum)	185	271
Number of fun-expression (avg)	5.138	4.839
Number of fun-expression (max)	20/18/17	31/26/22

6.1. táblázat. A *RefactorErl* két különböző verzióján mért bonyolultsági értékek

Más részről, az eszköz funkcionalitásának növekedése miatt, valamint a generált forráskód elemzők bevezetésével a függvények száma jelentős mértékben megnövekedett. Mivel funkcionális programról van szó, a függvények számának növekedésével a rekurzív ágak száma, valamint a függvény ágak száma is jelentős mértékben nőtt, ahogy ezt a mérési eredmények is alátámasztják.

A különböző kifejezések (főként az elágazások) beágyazottságának száma kis mértékben csökkent, ami szintén a generált kód eredménye.

6.2. megjegyzés. *A forrásszöveget automatikusan előállító program a beágyazott kifejezések helyett a több ágból álló függvényeket részesíti előnyben, míg a programozók sokszor inkább az előbbieket választják.*

A szoftver forrásszövegének átalakítása során kitűzött célok nagy része megvalósult. A modularizáció a modulok közti kapcsolatok számát csökkentette, a modulokon belüli kapcsolatok erőssége növekedett. A modularizáció hatására a programkód stílusa, olvashatósága, alakíthatósága javult, míg a bonyolultsága csökkent.

Mindezek azért fontosak a számunkra, mert a táblázatban található értékek ugyanezen állításokat támasztják alá és ezzel párhuzamosan igazolják, hogy a bonyolultsági mértékek jól mérnek, vagyis tükrözik a forrásszöveg változásait.

6.5. Transzformációs szkriptnyelv validálása

Annak érdekében, hogy a transzformációs nyelvről a fejezetben kimondott állításainkat igazoljuk, számos optimalizáló szkriptet futtatunk, majd ellenőriztük, hogy azok hozzák-e az elvárt eredményeket. A fejezetben ezek közül egyet, a 6.5 példában szereplő és az optimalizálással foglalkozó 4 fejezetben szintén bemutatott szkriptet lefuttattuk a 6.6 forrásszövegre. Ahogy láthatjuk, az ott leírt, elvárt eredmény, és az itt bemutatott tényleges kimenete a programnak nem sokban tér el.

```
optimize
  extract_fun
where
  max_depth_of_cases > 1
 and
  number_of_function < 10
limit 2
```

6.5. ábra. Beágyazottságot csökkentő szkript

Az eltérés oka az, hogy a *RefactorErl* a függvény neveket generálta, mi viszont korábban megadtuk azt. A forrásszöveg, amely tartalmazta az eredeti, három mélységben beágyazott *case* kifejezéseket, kibővült két új, számos kompenzáció során létrejövő paraméterekkel ellátott függvénnyel, amelyek tartalmazzák a kiemelt kifejezéseket.

A forráskódot a transzformációs lépések lefuttatása után lefordítottuk, hogy igazoljuk a transzformációs lépések jelentésmegőrző viselkedését. A példaprogram működött és a transzformáció előttivel azonos eredményeket produkált.

```
-module(opt).
-export([f/1]).

f({A, B})->
case A of
  send -> case B of
       {Pid, Data} -> case Pid of
                        {pid, P} ->
                                  P ! Data;
                                _ ->
                                  Pid ! B
                      end;
                  _ -> null ! B
         end;
       _ -> mod:wait()
end.
```

6.6. ábra. Átalakításra váró forrásszöveg

```
-module(opt).
-export([f/1]).

f({A, B})->
case A of
        send -> opt_65(B);
           _ -> mod:wait()
end.

opt_65(B) ->
    case B of
        {Pid, Data} ->
            opt_68(B, Data, Pid);
        _ -> null ! B
    end.

opt_68(B, Data, Pid) ->
    case Pid of
        {pid, P} -> P ! Data;
        _ -> Pid ! B
    end.
```

6.7. ábra. A szkripttel átalakított forrásszöveg

Az első teszt elvégzését követően a kapott eredményfájlra meghívtuk a transzformációkat végrehajtó algoritmus *undo* műveletét, hogy visszakapjuk az eredeti programot, majd egy kibővített szkriptet futtattunk az újabb tesztek elvégzéséhez. A szkriptet a 6.8. ábrán találjuk.

```
optimize
   extract_fun (exprtype, case_expr)
where
  f_mcCabe > 10
   and
  f_max_depth_of_structures > 2
   and
  f_max_depth_of_cases > 1
limit 5;

optimize
   move_fun (targetmod, 'funlib')
where
  f_external_calls > 1,
  f_internal_calls < 1,
  f_external_calls_from > 1,
  f_internal_calls_from < 1,
  m_number_of_fun > 1
limit 2;

optimize
   rename_fun (prefix, 'lib_')
where
  module == 'funlib'
limit 5;

optimize
  tuple_funpar
where
  f_number_of_vars > 1
limit 1;

optimize
  introduce_record
where
  m_name == 'funlib',
  f_number_of_vars > 0
limit 1
```

6.8. ábra. Modulszervező szkript

A 6.8 szkript feladati a következők: Megkeresi az összes elemzett, és a $\mathcal{SG}$ szemantikus gráfban tárolt modul függvényei közül azokat, amelyekben a *case* kifejezések beágyazottsága nagyobb, mint kettő, majd kiemeli ezeket az *extract_fun* transzformáció alkalmazásával. Ezt a lépést maximum ötször végzi el.

A következő lépésben minden olyan függvényt a *funlib* nevű modulba mozgat, amelyek a feltételében leírtaknak megfelelnek, vagyis nem hívják más modul függvényeit, de hívják őket más modulokból legalább egyszer (ezek mindenképpen exportált függvények).

A *limit* szekció ezeket a lépéseket maximum egy alkalommal engedi lefuttatni, mivel a mozgatások miatt a cél modul is fókuszba kerülhetne, és legrosszabb esetben is fölösleges mozgatásokat végeznénk el. Amennyiben nem létezik a célmodul, létrehozza azt a megadott néven, majd hozzáadja a szemantikus gráfhoz.

A mozgatásokat követően a következő szekció minden, a *funlib* modulban található függvényt ellátja a *lib_* előtaggal. Itt a *limit* résznek nincs különösebb jelentősége.

Az átnevezések után a típusokkal való bővíthetőség és a hatékonyabb átalakíthatóság érdekében alkalmazni kell a *tuple_funpar* transzformációt minden olyan függvényen, ahol a paraméterek száma nagyobb nullánál, majd ezt követően az *introduce_record* transzformációt. Ez a lépés rekorddá alakítja a *funlib* modul függvényeinek korábban n-esekbe rendezett formális paraméterlistáját (ahol a paraméterek száma nem nulla).

Természetesen a transzformáció elvégzi a megfelelő kompenzációkat, létrehozza a rekordokat, kompenzációs makrókat (részletesen lásd: 3. fejezetben), és ha szükséges, akkor a fejléc fájlokat a közös rekordokat használó, de más modulokban található függvények számára. Ennek a szekciónak is fontos a *limit* része, mivel a rekord paraméter is egy paraméternek számít, és így a *number_of_vars* feltétel miatt a szkript a végtelenségig ágyazná egymásba a rekord definíciókat.

Irodalomjegyzék

[1] Roland Király Results of complexity metric based automatic source code transformations. *Annales Mathematicae et Informaticae 42 (2013) (accepted)*

[2] Tóth, M., Bozó, I., Horváth, Z., Kitlei, R., Király, R., Horpácsi, D., and Kőszegi, J.: *RefactorErl: a source code analyser and transformer tool* Poster at the High Speed Network Workshop 2011, Budapest, Hungary, May 2011

[3] Roland Király, Róbert Kitlei. *Metric Based optimalization of functional source code.* Annales Mathematicae et Informaticae 37 (2011) Pages: 59-74

[4] István Bozó, Dániel Horpácsi, Zoltán Horváth, Judit Kőszegi, Roland Király, Róbert Kitlei, Máté Tejfel, Melinda Tóth,. *Haladó technológiák szoftverrendszerek forráskódú elemzésére A RefactorErl hatékonyságának és felhasználói felületének továbbfejlesztése* Az Ericsson Magyarország Kft megbízásából és támogatásával a KMOP-1.1.2-08/1-2008-0002 projekt keretében az ERFA támogatásával Tech report 2011 Ericsson Hungary

[5] Király, R., Kitlei R.: *Complexity measurments for functional code* 8th Joint Conference on Mathematics and Computer Science (MaCS 2010) refereed, and the proceedings will have ISBN classification July 14-17, 2010

[6] Király, R., Kitlei R.: *Implementing structural complexity metrics in Erlang.* '10 ICAI 2010 – 8th International Conference on Applied Informatics to be held in Eger, Hungary January 27-30, 2010

[7] Király, R. and Kitlei, R.: *Implementing structural complexity metrics for Erlang* Poster on the 8th International Conference on Applied Informatics, ICAI 2010, 2010

[8] Zoltán Horváth, László Lövei Tamás Kozsik, Roland Király, Melinda Tóth, Róbert Kitlei, Dániel Horpácsi, István Bozó. *Extended semantic queries on Erlang programs and comprehensive testing of RefactorErl.* Tech. Report 2010. Ericsson Hungary 2010.

[9] Király Roland, *Funkcionális programozási nyelvek* EKF TTK TAMOP412 2010 120 oldal.

[10] Zoltán Hernyák, Roland Király. *Teaching programming language in grammar schools.* Annales Mathematicae et Informaticae 36 (2009) Pages: 163-174

[11] Horváth, Z., Lövei, L., Kozsik, T., Kitlei, R., Víg, A., Nagy, T., Tóth, M., and Király, R.: *Modeling semantic knowledge in Erlang for refactoring.* In Knowledge Engineering: Principles and Techniques, Proceedings of the International Conference on Knowledge Engineering, Principles and Techniques, KEPT 2009, volume 54(2009) Sp. Issue, Studia Universitatis Babeş-Bolyai, Series Informatica, pages 7–16, Cluj-Napoca, Romania, Jul 2009

[12] Lövei, L., Tóth, M., Horváth, Z., Kozsik, T., Király, R., Kitlei, R., Bozó, I., Hoch, C., and Horpácsi, D.: *Reengineering legacy Erlang code by refactoring.* Central Europen Functional Programming Summer School, May 2009.

[13] Thanassis Avgerinos, Konstantinos F. Sagonas *Cleaning up Erlang code is a dirty job but somebody's gotta do it.* Erlang Workshop 2009: 1-10

[14] Konstantinos F. Sagonas, Thanassis Avgerinos *Automatic refactoring of Erlang programs.* PPDP '09 Proceedings of the 11th ACM SIGPLAN conference on Principles and practice of declarative programming 2009: 13-24

[15] Zoltán Horváth, László Lövei Tamás Kozsik, Roland Király, Melinda Tóth, Róbert Kitlei, Dániel Horpácsi, István Bozó. *Complexity Metrics and simple semantic queries for Erlang.* Report Ericsson Hungary 2009.

[16] Tamás Kozsik, Zoltán Csörnyei, Zoltán Horváth, Roland Király, Róbert Kitlei, László Lövei, Tamás Nagy, Melinda Tóth, Anikó Víg Use Cases for Refactoring in Erlang. *In Central European Functional Programming School, volume 5161/2008, Lecture Notes in Computer Science, pages 250–285, (2008)*

[17] R. Kitlei, L. Lövei, M Tóth, Z. Horváth, T. Kozsik, T. Kozsik, R. Király, I. Bozó, Cs. Hoch, D. Horpácsi. *Automated Syntax Manipulation in RefactorErl.* 14th International Erlang/OTP User Conference. Stockholm, November 13, 2008.

[18] Lövei, L., Hoch, C., Köllő, H., Nagy, T., Nagyné-Víg, A., Kitlei, R., and Király, R.: *Refactoring Module Structure* In 7th ACM SIGPLAN Erlang Workshop, 2008

[19] Zoltán Horváth, László Lövei, Tamás Kozsik, Róbert Kitlei, Anikó Nagyné Víg, Tamás Nagy, Melinda Tóth, and Roland Király. *Building a Refactoring Tool for Erlang.* In K. Mens, M. van den Brand, A. Kuhn, H.M. Kienle, and R. Wuyts, editors, 1st International Workshop on Academic Software Development Tools and Techniques, 2008. 11 pages.

[20] Lövei, L., Hoch, C., Köllő, H., Nagy, T., Nagyné-Víg, A., Horpácsi, D., Kitlei, R., and Király, R.: *Refactoring Module Structure* In Proceedings of the 7th ACM SIGPLAN workshop on Erlang, pages 83–89, Victoria, British Columbia, Canada, Sep 2008.

[21] Lövei, L., Hoch, C., Köllő, H., Nagy, T., Nagyné-Víg, A., Horpácsi, D., Kitlei, R., and Király, R.: *Refactoring module structure in.:Proceedings of the 7th ACM SIGPLAN workshop on ERLANG Columbia, Canada, (2008)*

[22] Zoltán Horváth, Zoltán Csörnyei, Roland Király, Róbert Kitlei, Tamás Kozsik, László Lövei, Tamás Nagy, Melinda Tóth, and Anikó Víg.: Use cases *for refactoring in Erlang, To appear in Lecture Notes in Computer Science, (2008)*

[23] R. Kitlei, L. Lövei, M Tóth, Z. Horváth, T. Kozsik, T. Kozsik, R. Király, I. Bozó, Cs. Hoch, D. Horpácsi.: Automated Syntax Manipulation in RefactorErl. *14th International Erlang/OTP User Conference. Stockholm, (2008)*

[24] Lövei, L., Hoch, C., Köllő, H., Nagy, T., Nagyné-Víg, A., Kitlei, R., and Király, R.: Refactoring Module Structure *In 7th ACM SIGPLAN Erlang Workshop, (2008)*

[25] Zoltán Horváth, László Lövei Tamás Kozsik, Anikó Nagyné Víg, Tamás Nagy, Roland Király, Melinda Tóth, Róbert Kitlei, Dániel Horpácsi, Hanna Köllő, Krisztián Tóth, Csaba Hoch.: *Erlang programok modulszerkezetének refaktorálása, Technical report (2008)*

[26] Horváth, Z., Lövei, L., Kozsik, T., Kitlei, R., Víg, A., Nagy, T., Tóth, M., and Király, R.: Building a refactoring tool for Erlang *In Workshop on Advanced Software Development Tools and Techniques, WASDETT 2008, (2008)*

[27] Huiqing Li, Simon Thompson, György Orosz, Melinda Tóth.: Refactoring with wrangler, updated: data and process refactorings, and integration with Eclipse *Proceedings of the 7th ACM SIGPLAN workshop on ERLANG (2008)*

[28] Lövei, L., Horváth, Z., Kozsik, T., Király, R., Víg, A., and Nagy, T.: *Refactoring in Erlang, a Dynamic Functional Language* In Proceedings of the 1st Workshop on Refactoring Tools, pages 4546, Berlin, Germany, 2007 extended abstract

[29] László Lövei, Zoltán Horváth, Tamás Kozsik, Roland Király, and Róbert Kitlei. *Static rules of variable scoping in Erlang.* In Emőd Kovács, Péter Olajos, and Tibor Tómács, editors, Proceedings of the 7th International Conference on Applied Informatics, volume 2, pages 137-145, 2008. rev: Zbl pre05662517.

[30] Tamás Kozsik, Zoltán Csörnyei, Zoltán Horváth, Roland Király, Róbert Kitlei, László Lövei, Tamás Nagy, Melinda Tóth, and Anikó Víg. *Use Cases for Refactoring in Erlang.* In Central European Functional Programming School (The Second Central European Summer School, CEFP 2007, Cluj, Romania, June 23-30, 2007), Revised Selected Lectures, volume 5161 of Lecture Notes in Computer Science, pages 250-285. Springer Berlin/Heidelberg, 2008. rev: Zbl 1170.68414, DBLP

[31] László Lövei, Zoltán Horváth, Tamás Kozsik, and Roland Király. *Introducing records by refactoring.* In Proceedings of the 2007 ACM SIGPLAN Erlang Workshop, pages 18-28. ACM Press, 2007.

[32] Lövei, L., Horváth, Z., Kozsik, T., Király, R., Víg, A., and Nagy, T.: Refactoring in Erlang, a Dynamic Functional Language *In Proceedings of the 1st Workshop on Refactoring Tools, pages 45–46, Berlin, Germany, extended abstract, poster (2007)*

[33] T. Kozsik, Z. Horváth, L. Lövei, T. Nagy, Z. Csörnyei, A. Víg, R. Király, M. Tóth, R. Kitlei.. Refactoring Erlang programs. *CEFP'07, Kolozsvár (2007)*

[34] Zoltán Porkoláb, Ádám Sipos, Norbert Pataki, Structural Complexity Metrics on SDL Programs. *Computer Science, CSCS 2006, Volume of extended abstracts, (2006)*

[35] Horváth, Z.: Elosztott funkcionális programok helyessége (Verification of Distributed *Functional Programs) Project report, OTKA. http://www.otka.hu, (2006)*

[36] Csörnyei Zoltán *Fordítóprogramok* Typotex Kiadó, Budapest, 2006.

[37] Ryder, C., Thompson, S. *Software Metrics: Measuring Haskell,* In Marko van Eekelen and Kevin Hammond, editors, Trends in Functional Programming (September 2005)

[38] Ryder, C. Software Measurement for Functional Programming, *PhD thesis, Computing Lab, University of Kent, Canterbury, UK 2004)*

[39] Fóthi Á., Nyéki-Gaizler J., Porkoláb Z. The Structured Complexity of Object Oriented Programs *Computers and Mathematics with applications, (2002)*

[40] Zoltán Porkoláb Programok Strukturális Bonyolultsági Mérőszámai. *PhD thesis Dr Töke Pál, ELTE Hungary, (2002)*

[41] Frank Simon, Frank Steinbrückner, Claus Lewerentz *Metrics based refactoring* IEEE Computer Society Press 2001 30 - 38,

[42] Claus Lewerentz, Frank Simon *A Product Metrics Tool Integrated into a Software Development Environment* Object-Oriented Technology (ECOOP'98 Workshop Reader), LNCS 1543 Springer-Verlag 256–257

[43] Klaas van den Berg.: Software Measurement and Functional Programming, *PhD Thesis University of Twente (1995)*

[44] Fóthi Á., Nyéki-Gaizler J. On The Complexity of Object-Oriented Programs *in Proc. of the 3rd Symp. on Programming Languages and Software Tools Kaariku, Estonia, (1993)*

[45] Howatt, J.W. and Barker. A.L. *Rigorous Definition and analisys of Program Complexity Measures: An Example Using Nesting* The Journal of systems and Software vol.10, pp.139-150 (89')

[46] Piwowarsky, P. A Nesting Level Complexity Measure. *ACM Singplan Notices 17(9) pp.44-50 (1982)*

[47] Henry S., Kafura D. *Software Structure Metrics Based of Information Flow*, IEEE Trans. Software Engineering, vol.7, pp.510-518 (81')

[48] McCabe T. J. A Complexity Measure, *IEE Trans. Software Engineering, SE-2(4), pp.308-320 (1976)*

[49] Halstead, M. H. *Natural laws controlling algorithm structure* SINGPLAN Notice, vol.7. pp19-26 (72')

[50] Bohem, B. W. *Software and its impact:A quantitative assessment*, Datamation vol.19, pp.48-59 (73')

[51] The IBM United States
http://www.ibm.com/us/en/

[52] Erlang - Dynamic Functional Language
http://www.erlang.org

[53] Eclipse Foundation
http://www.eclipse.org/